DISASSEMBLER USING HIGH LEVEL PROCESSOR MODELS

By

NIHAL CHAND JAIN

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY, KANPUR

EMBER, 1998

Disassembler using High Level Processor Models

A Thesis Submitted
in Partial Fulfillment of the Requirements
for the Degree of
Master of Technology

by
Nihal Chand Jain

to the

Department of Computer Science & Engineering
Indian Institute of Technology, Kanpur

December, 1998

Certificate

Certified that the work contained in the thesis entitled " *Disassembler using High Level Processor Models*", by Mr.*Nihal Chand Jain*, has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

(Dr. Rajat Moona)
Associate Professor,
Department of Computer Science & Engineering,
Indian Institute of Technology,
Kanpur.

December, 1998

Abstract

The design of a high performance system requires an integrated environment to simulate and analyze the performance of various design alternatives. In this thesis, we have developed a generic disassembler for an integrated environment where Sim-nML acts as the specification language for processor performance model. The Sim-nML, an extension of nML machine description formalism, is a simple, elegant and powerful language to model machine behavior at instruction level. As part of the thesis work, we have designed an intermediate representation (IR) for processor specification written in Sim-nML language. The IR is simple and facilitates the development of various tools such as assembler, compiler back-end generator, instruction set simulator, trace generator etc. based on the processor specification. A tool, IR-Generator, is developed which takes a processor specification written in Sim-nML language and produces it in the intermediate representation. Further, a Generic Symbolic Disassembler is developed which takes the intermediate representation of a processor and a relocatable binary file in ELF format as input and produces an equivalent program in assembly language of the processor. The disassembler is generic enough to be used for all type of processors.

Acknowledgments

I am grateful to my thesis supervisor, Dr. Rajat Moona, who guided me at every stage of this project with his valuable suggestions, whose qualities have attracted me a lot. His ideas have been of great help in exploring the areas which otherwise would have been impossible. I thank the Almighty for giving such a brotherly figure as my guide.

This work is done as a part of the ongoing research in Cadence Research Center at IIT Kanpur. I express my gratitude to Cadence India Ltd. for their enduring support to this work. Apart from the ample financial support provided by the fellowship, it has been a source of personal pride and motivation to be called a "Cadence Fellow".

I am also greatly indebted to Dr. Deepak Gupta and Dr. Sanjeev Agarwal for their guidance and support throughout my work. I express my heart-felt thanks to all the faculty members for teaching the principles in most exciting and enjoyable way.

I am greatly indebted to my seniors specially Atul and Kshitiz for helping me out in crucial situations. I would also like to thank V.Rajesh, Subhash and Shishir who has helped me with lots of ideas throughout the work. I thank all my MTech97 classmates especially Professor (Zade), Bepari, Kapil, Atul, Manoj, Anna, Anjali, Uma, Prasad, Srikar, Gopi, Girish, Prasanna, Kousik and Major Ajay, for being affectionate and the source of inspiration for me. My gratitude goes to all of my MTech97 batchmates, who made my stay in Hall-V, IITK a memorable one. I acknowledge the MTech98 batch for their exciting company. I wish I could express my thankfulness to all my old friends for their love, support and encouragement.

I thank my parents, my brothers, for their love and affection I have been receiving. I am grateful to all of them for their efforts in building my career. Finally, I would also like to thank God for being kind to me and driving me through this journey.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The design of a high performance system requires complex software tools. Designers use powerful and generic modeling tools to evaluate many alternative implementations. In addition, designers need hardware and software codesign and other trade-offs at early stages of the system design to keep the development cost down. Therefore, system designers need an integrated environment which allows them to simulate and analyze the performance of various design alternatives.

In this thesis, we have used Sim-nML language[14] which is primarily an extension of the nML language[1] for processor modeling and designed a generic processor independent symbolic disassembler. For this purpose, we have also designed an intermediate representation (IR) for the processor specification written in the Sim-nML language. The IR is simple but powerful enough to facilitate the development of various tools such as assembler, compiler back-end generator, instruction simulator etc. based on the processor specification. We have designed a tool, IR-generator, which takes a processor specification in the Sim-nML language and provides the intermediate representation of the processor specification as output. The generic symbolic disassembler takes the intermediate representation and a relocatable binary file of a processor and provides the corresponding assembly language program as output.

1.1 Motivation

A processor model provides means to facilitates hardware and software codesign and coanalysis early in the system design process. To model the candidate application and processor model interaction. a systematic design process is required. A systematic design process starts with selecting the application and involves writing a model that measures the performance of the system, testing the system, analyzing the results and refining the model to enhance performance. In this process, the model undergoes several changes till the desired performance is achieved. This approach requires to have an environment where changes to the design are made at one place and the corresponding changes in other tools are automated. Such an integrated environment can incorporate the model changes and validation rapidly.

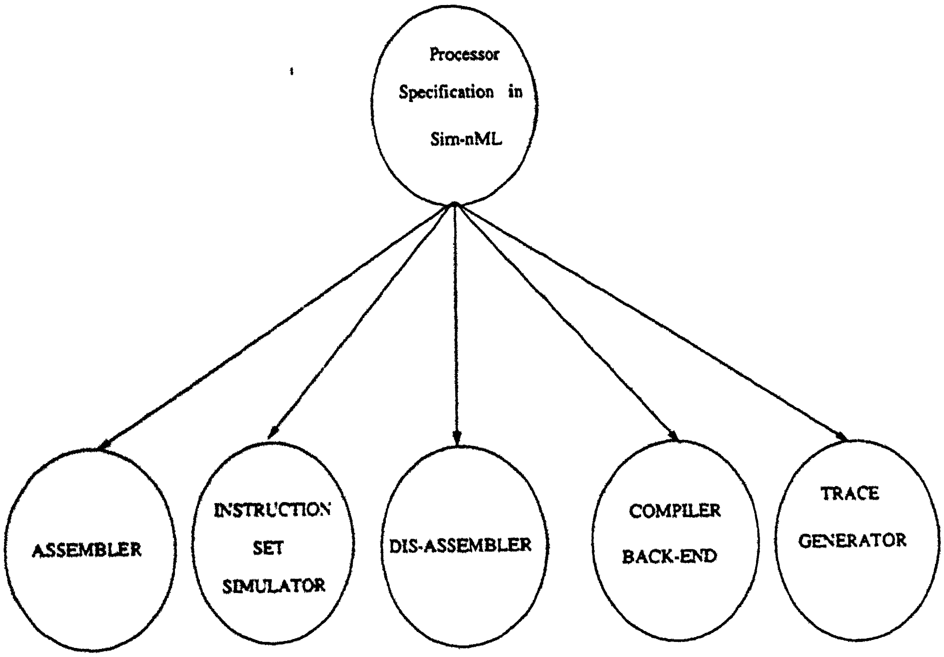

Figure 1: System Overview

In this thesis, we are discussing the design of an integrated environment where Sim-nML language[14] acts as the specification language for processors in a generic way. The processor specification written in Sim-nML language can be used to generate various processor specific tools such as compiler back-end generator, assembler, disassembler, trace generator, instruction set simulator (ISS) etc. The ISS can simulate the execution of a binary program using the processor specification. Therefore, the ISS helps in performance modeling. A compiler back-end generator integrated

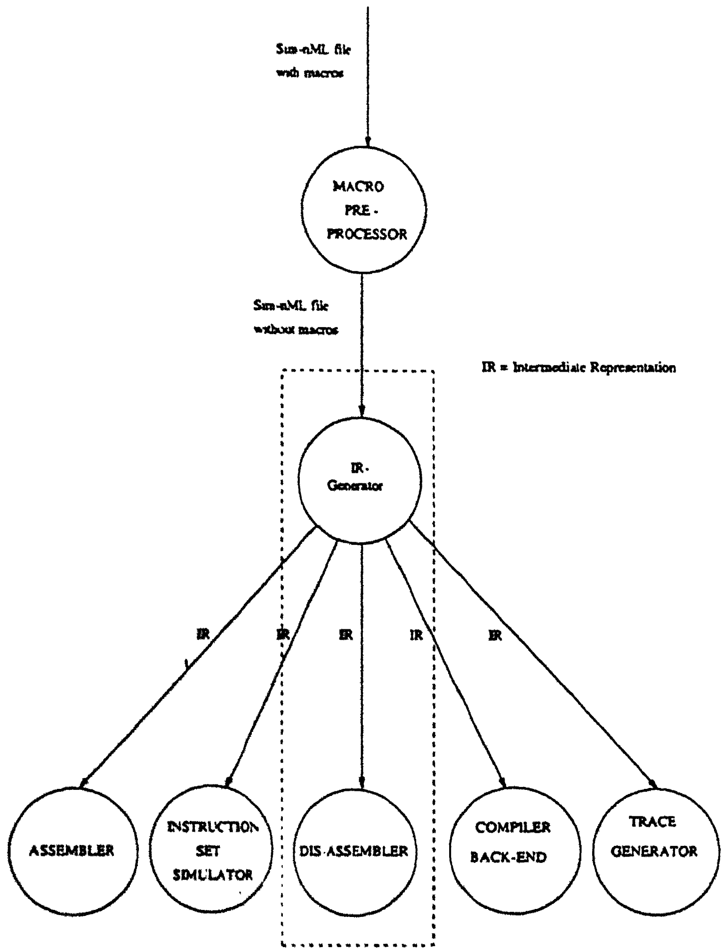

Figure 2: System Overview with IR

with a compiler front-end can be used to generate a complete compiler for the same processor. Thus an integrated environment, as shown in figure 1, is designed with Sim-nML as the specification language to automate the generation of various processor specific tools. While it is convenient to describe processor models in Sim-nML, it is not so convenient for the tools to use Sim-nML specification directly as input due to hierarchical description formalism of Sim-nML. Further, the direct usage requires all tools to duplicate the effort in processing of processor specifications. This motivated us to design a simple intermediate representation (IR) for Sim-nML language specification so that the design of various tools is simplified. The system design can be viewed as shown in figure 2 with integration of the IR and previous view. In embedded applications, it is helpful to study the algorithm used by the application program and then change it to suit the specific needs. However, it is convenient to operate at assembly language level rather than at the binary executable level. This

motivated us to design a symbolic generic disassembler to translate a relocatable binary code to its assembly language counterpart. In the symbolic disassembly, *symbols* are used to refer to the locations and functions rather than the absolute addresses in the assembly language program. The disassembler developed in this thesis uses the processor models specified in the IR (*thus it is generic*). The assembly language program helps in extracting a lot of hidden information and further improvements or analysis can be done.

1.2 Overview of Related Work

Performance modeling of a system is a growing area and a lot of research has been pursued in this area. These previous works has resulted in a set of performance modeling tools using different languages for processor specification.

VHDL[9] is an expressive language with full hierarchy and configurations that allow development and application of highly configurable and flexible models. There are wealth of VHDL-based modeling tools as described in various works[9, 8].

SLED[6], a Specification Language for Encoding and Decoding, is used for abstract, binary and assembly-language representation of machine instructions. SLED is suitable for describing both CISC and RISC machines. Processor representation for MIPS, SPARC, Alpha, Pentium, PowerPC and Motorola 68000 are also written in SLED and a toolkit, the New Jersey Machine-Code (NJMC) Toolkit, is implemented to help programmers write applications that process machine code—assemblers, disassemblers, code generators, tracers, profilers, and debuggers. A Disassembler for SPARC is also implemented in the NJMC using SLED as processor specification language[7].

Visualization based Microarchitecture Workbench (VMW)[13] is an infrastructure which facilitates the specification of instruction set architecture and microarchitecture of a machine in concise manner. VMW provides all necessary infrastructure software to the designer, including generic simulation software, visualization support software and graphical user interface software. VMW automatically integrates the machine specification and infrastructure software to generate a customized performance simulator based on the trace-driven simulation approach. Thus VMW provides a powerful environment for modern superscalar processor design.

Trimaran System[11] is an integrated compilation and performance monitoring infrastructure for Instruction Level Parallel (ILP) architectures. The ILP architecture (HPL-PD), parameterized by a machine description, allows the user to experiment with different machines. The HPL-PD architecture supports novel features such as predication, control and data speculation and compiler controlled management of the memory hierarchy. A cycle-level HPL-PD simulator provides a detailed simulation environment to get various information. The information is used for profile-driven optimization and for validation of new optimization. The machine description is specified in a high level textual language HMDES[12]. A compiler front-end (IMPACT) and a compiler back-end (Elcor), parameterized by the machine description, together provides experimentation for new ILP architectures and the compiler modules needed to generate high-performance code for these architectures. The modular Elcor uses an intermediate representation throughout its module which enable the construction and insertion of new compilation modules into the compiler in a easy way.

Other than these complete machine simulation environments, many performance models exist for analyzing the individual components such as processors, caches etc. A Framework for Statistical Modeling of Super-scalar Processor Performance is discussed in [10]. Performance Estimation for Real-Time Distributed Embedded Systems is discussed in [15]. An ISS (Instruction Set Simulator)[5] is developed to simulate an architecture of a processor which is defined through "templates". Further, a performance simulator[5] is implemented using traces from the ISS as input which has been used to evaluate the Ultra-SPARC-compatible architecture. A cycle accurate model of Ultra-SPARC processor is written in C++ to verify the processor by cross checking the RTL model at run time as well as to provide accurate performance estimates[3].

In the area of disassembling, several disassemblers have been implemented as listed in [17]. Among these, IDA Pro[2] is a disassembler based on FLIRT (Fast Library Identification and Recognition Technology) and can disassemble binary files for several processors.

1.3 Goals Achieved

In this thesis work, we aimed at developing an integrated environment for processor modeling using Sim-nML language for processor specification. The development of a complete integrated environment is in progress where other tools (i.e. simulator,

trace generator) are under development. The goals achieved in this thesis work are listed below.

- Intermediate Representation (IR) for Sim-nML language specification is designed which is simple but powerful enough to facilitate the design of various processor specific tools.
- IR-Generator is designed and implemented which takes a processor specification in Sim-nML language and provides an intermediate representation of the processor specification as output.
- Symbolic Generic Disassembler is designed and implemented which takes the intermediate representation and a relocatable binary file of a processor and provides corresponding assembly language program of the processor as output.

Work done in this thesis is also outlined in the figure 2.

1.4 Organization of Report

The rest of the thesis is organized as follows. In chapter 2, we describe the design of the intermediate representation and the implementation of the IR-Generator after giving an overview of the Sim-nML language. In chapter 3, we describe the design and implementation of the symbolic generic disassembler. Finally we conclude in chapter 4 and provide the results. We also enumerate possible future work in this area. Context-free grammar of the Sim-nML language is listed in Appendix A. Detail format of the intermediate representation is given in Appendix B. Lastly, user's manuals for the IR-Generator and the disassembler are given in Appendix C.

Chapter 2

Intermediate Representation of Processor Models

One part of this thesis involves the development of Intermediate Representation (IR) of the processor model. We developed a tool, IR-Generator, which takes a processor specification written in Sim-nML language as input and produces corresponding intermediate representation of processor specification as output. In order to have intermediate representation usable by all front-end tools such as disassembler, assembler, simulator etc., certain goals were setup behind the design of the IR as listed below.

- The IR should be as simple as possible.
- The IR should not lose any useful information which is available in original input of Sim-nML specification.
- The IR should not have any unnecessary or redundant information.
- The IR should be easy to understand as well as to use.
- It should be easy and efficient to retrieve the required information from the IR.
- The IR should be flexible and extensible.
- The IR should facilitate the design of various processor specific tools such as assembler, disassembler, simulator, trace generator, compiler back-end generator etc.

Before discussing the IR in detail, it is necessary to understand the structure of the input. Sim-nML work by V.Rajesh [14] is primarily an extension of nML[1] (designed by Markus Freerick). Here we will discuss Sim-nML in brief for better understanding of our work. More information about Sim-nML can be found in relevant literature [14, 4].

2.1 Sim-nML Language

Sim-nML[14] is an extensible formalism targeted for describing arbitrary single processor computer architecture. It facilitates the description at instruction set level and hides the implementation details. In Sim-nML, the instruction set is enumerated by an *attribute grammar*[1]. The semantic action of an instruction is composed of fragments that are distributed over the whole specification tree, i.e. the common behavior of a class of instructions is captured at the top level of the tree and the specialized behavior of sub-classes is captured at the subsequent lower levels.

2.1.1 Sim-nML Grammar

Sim-nML grammar has a fixed start symbol namely *instruction* and two kind of productions namely, *or-rule* which looks like,

```
op n0 = n1 | n2 | n3 | ...
```

and *and-rule* which looks like,

```
op n0 ( p1 : t1, p2 : t2, ... )
a1 = e1
a2 = e2
...
```

[1]An attribute grammar is a context free grammar in which for each non-terminal a fixed set of attributes and for each production a set of semantic rule is given. In Sim-nML grammar, all non-terminals have to have derivations. So, we don't differentiate between productions and non-terminals.

```
let REGS = 4
type long = card(32)
type index = card(REGS)

reg R[2**REGS,long]
reg PC[1,long]
reg AC[1,long]

mode REG(i:index)=R[i]
syntax = format("R%d",i)
image = format("%4b",i)

op instruction(x:instr_action)
action = { PC = PC + 4;
           x.action; }
syntax = x.syntax
image = x.image

op instr_action = move | store

op move(src:REG)
action = { AC = src; }
syntax = format("load %s",src.syntax)
image = format("0010 %b",src.image)

op store(src:REG)
action = { src = AC; }
syntax = format("store %s",src.syntax)
image = format("0011 %b ",src.image)
```

Figure 3: Sim-nML Specification for a Simple Processor

`n0, n1, n2, n3,`... are non-terminals and each `ti` is a token. Each `ai` is an attribute name and `ei` is its definition. The `pi` are names of the parameters used in the attribute definitions.

Sim-nML grammar pre-defines some attributes namely *syntax*, *image*, *action*, *uses*, *volatile*, *alias*, and *init*. The *syntax* attribute describes the textual syntax of the instruction. The *image* attribute describes the binary coding of the instruction. The *action* attribute describes the semantics of an instruction. The *uses* attribute is

used to describe the resource usage model and the control flow of an instruction. The *volatile*, *alias* and *init* attributes are valid for memory variables. The *init* attribute is used to assign initial values to memory variables while *volatile* attribute is used to define the volatile name of the memory.

The Sim-nML description in figure 3, is that of a simple machine with two instructions, the `load` instruction which is used to load accumulator `AC` with the contents of a register specified by an argument, and the `store` instruction which is used to store the value of the accumulator `AC` to the register specified by an argument. The register `PC` has special semantics and points to the next-to-be-executed instruction.

In the most processors, addressing modes and instructions are orthogonal to each other. Therefore, describing an instruction with each of the possible addressing modes explode the size of the description. Therefore, Sim-nML separates addressing mode description as register addressing mode is described in figure 3 with declaration of *mode-rule* `REG`.

The Sim-nML also supports resource and exception declaration which are useful for resource usage model. In addition, Sim-nML supports macros and declarations for types and constants. This enhances the clarity of the description. In appendix A, Sim-nML grammar is given in detail.

The Sim-nML formalism helps in describing the processor concisely and precisely. Thus Sim-nML description of a processor can be used as input to various tools such as assembler and disassembler generators, compiler back-end generators and general purpose instruction set simulators.

2.2 Design of an Intermediate Representation

A processor specification in Sim-nML language is a human readable text file. Several constructs are provided in Sim-nML to enhance the clarity and readability of the description. In order to retrieve the desired information from such a description, a tool needs to perform parsing of input, variable substitution etc. An intermediate representation helps in reducing such extra burden on the tool. Thus we need an intermediate representation keeping previously mentioned goals in mind. In this section, we will discuss the design of the IR in detail.

2.2.1 Simplification of Information by Substitution

Sim-nML language allows the constant definition using `let-specification` (i.e.`let REGS = 4`). In Sim-nML specification file, wherever a constant is referenced, its value is substituted in the IR. For example, value of the constant `REGS`, i.e. 4, is substituted where-ever `REGS` is used in the example given in figure 3. Thus constants are not referenced in the IR of the processor specification. Therefore all such constant declarations can be eliminated from the IR. However some constant might be used by the tools i.e. constant like `byte_order` may be used by tools to define the byte ordering of a processor. As it is difficult to guess what all constant definitions might be used by all such tools, it was decided to retain information about all constant declarations in the IR even if these are not referenced anywhere.

Sim-nML language has some basic data types and allows new data type definitions using basic data types and previously defined user data types. Since all user defined data types can be built using only basic data types, all variables are redefined with only basic data types in the IR. Thus all user defined data type declarations are of no use and are eliminated from the IR. For example, parameter `i` in *mode-rule* `REG` is redefined with data type `card(4)`. Now type definition `index` is eliminated from the IR.

Sim-nML allows macro declarations (macro name and macro definition) in the processor specification to save user's effort in writing it. These macro declaration may have parameters and may use macros within the macro definition. Wherever a macro name is used in the specification, corresponding macro definition is substituted in the IR. Thus all macro declarations are eliminated from the IR.

There are some other constructs in the Sim-nML which are simplified in the IR. For example, in the Sim-nML, all memory variables, *op-rules*, attribute names, parameter names in *and-rule* etc. are given unique identifier names and everywhere corresponding identifier name is used for reference. As length of an identifier name is variable, it wastes a lot of processing time to retrieve the information about a particular identifier. Sim-nML also allows the use of some identifier name for *op-rules* even before they are defined. This necessarily requires a tool to do multiples passes over processor specification. Many of these identifiers are not significant at all (for example, parameter names). In the IR, all significant identifiers are assigned a unique integer key and all their references are replaced by the use of the corresponding key.

It simplifies the information retrieval from the IR. The mapping between the key and the identifier is also provided in the IR (though the tools may never need to refer to these).

2.2.2 Simplifying the Hierarchy

In Sim-nML, information about an instruction is composed of fragments that are distributed over the whole specification tree with root node named as *instruction*. To get information about one particular instruction, a complete path from root node to a leaf node is traversed with proper parameter substitution at all levels of the tree. If all such paths are traversed, then information about all possible instructions are obtained. This process is called flattening of the tree. In the IR, information about the instructions are flattened using two different algorithms.

First algorithm performs flattening of all *or-rules* and is described in figure 4. Basically, all references of any *or-rule* are eliminated from all the *or-rule* and *and-rule* definitions. Therefore, all *or-rule* definitions can be eliminated from the IR. But some *or-rule* definitions might be used by other tools. For example, if root node *instruction* itself is an *or-rule*, then information about all its children will be useful for the tools. Therefore, all *or-rules* resultant from the algorithm are stored collectively at one place in the IR, even if these *or-rules* will not be referenced anywhere.

Elimination of *or-rule* parameters from an *and-rule* definition results in generation of new *and-rules*. All attributes of the *and-rule* remain unchanged in the new *and-rules*. To make the IR compact, these new *and-rules* are treated as *sub-rules* of the original *and-rule*. All *sub-rules* of an *and-rule* are stored along with the *and-rule* in the IR. The references for the attributes in the *and-rule* are not duplicated for *sub-rules*.

Working of the algorithm can be understood with an example of Sim-nML program given in figure 5. Figure 6 explains the working of the algorithm on *or-rules*. Figure 7 shows the working of the algorithm on a particular *and-rule*.

2.2.3 Representation of Attribute Definition

In Sim-nML language specification, memory variables, *mode-rules* and *op-rules* declarations define attribute names and their definitions. The attribute definition is either an expression consisting of various operands and operator, or a sequence of statements

Algorithm 1 :

- For each *or-rule* R_i. do the following steps.
  1. For all child nodes of R_i, do the following step.
  2. If the child node is an *or-rule* C_i
     then replace the child node by all children node of C_i.
- For each *and-rule* A_i, do the following steps.
  1. For each parameter P_i of A_i, do the following step recursively.
  2. If P_i is an *or-rule* (say `R` where `R` has n-children namely $C_1, C_2 \ldots C_n$), then create n-new *sub-rules* and associate them with the *and-rule* A_i. In the i^{th} *sub-rule*, the parameter P_i is declared of type C_i.

Figure 4: Algorithm for Flattening of *or-rules*

separated by a semicolon. Each of these statements might be a simple assignment statement or a conditional statement or a function call or a use of an attribute from some *op-rule*. (Refer to appendix A for Sim-nML grammar)

For *syntax* and *image* attributes, definition is always an expression which evaluates to a string. In the IR, a record is stored for each *syntax* and *image* attribute definition. The record includes a string value corresponding to the expression. The string values are evaluated by algorithm 2 given in figure 8. Basically, the algorithm performs substitution of parameter values in the expression to evaluate the string value. However, the expressions also have references to parameters which can only be known at the run time of a tool. For example, *syntax* attribute definition of *mode-rule* REG has reference "%d" for parameter `i`. In the IR, a tuple "{X.Y.Z}" is used after a parameter reference such as "%d". Each tuple represents a parameter which can be converted using parameter reference. In the tuple, X denotes an *and-rule*, Y denotes a *sub-rule* and Z denotes a parameter number. Example in figure 9 provides the IR translation for the *syntax* attribute definitions of a few *and-rules* in the example given in figure 5.

The record holds another string called *dot-expression* as shown in figure 9. The *dot-expression* denotes the sequence of parameter substitution applied for calculating *syntax* and *image* attribute values. Each *dot-expression* contains a number of 2-tuples, each of type X.Y. All tuples except for the first one are put in parentheses. In

```
type index=card(2)
reg  PC[1,card(32)]
mode SHORT = MEM | REG

mode MEM(i:index)=M[R[i]]
syntax = format("(R%d)",i)
image = format("0%2b",i)

mode REG(i:index)=R[i]
syntax = format("R%d",i)
image = format("1%2b",i)

op instruction(x:instr_action)
syntax = x.syntax
image = x.image

op instr_action = alu_op | move_op
op alu_op(src:SHORT,dst:SHORT,aa:alu_action)
syntax = format("%s %s,%s",aa.syntax,src.syntax,dst.syntax)
image = format("1%b %b %b",aa.image,src.image,dst.image)

op alu_action = a_add | a_sub
op a_add()
syntax = "add"
image = "0"

op a_sub()
syntax = "sub"
image = "1"

op move_op = move | store
op move(src:SHORT,dst:SHORT)
syntax = format("move %s,%s",src.syntax,dst.syntax)
image = format("00 %b %b",dst.image,src.image)

op store(src:SHORT,dst:SHORT)
syntax = format("move %s,%s",src.syntax,dst.syntax)
image = format("01 %b %b",src.image,dst.image)
```

Figure 5: Sim-nML Program for a Hypothetical Processor

Before application of algorithm 1 :

```
op instr_action = alu_op | move_op
op alu_action = a_add | a_sub
op move_op = move | store
mode SHORT = REG | MEM
```

After application of algorithm 1 :

```
op instr_action = alu_op | move | store
op alu_action = a_add | a_sub
op move_op = move | store
mode SHORT = REG | MEM
```

Figure 6: Example of *or-rules* Flattening

Before application of algorithm 1:

```
op alu_op(src:SHORT, dst:SHORT, aa:alu_action)
```

After application of algorithm 1 :

```
op alu_op
        sub-rule 0 : src : REG, dst : REG, aa : a_add
        sub-rule 1 : src : REG, dst : REG, aa : a_sub
        sub-rule 2 : src : REG, dst : MEM, aa : a_add
        sub-rule 3 : src : REG, dst : MEM, aa : a_sub
        sub-rule 4 : src : MEM, dst : REG, aa : a_add
        sub-rule 5 : src : MEM, dst : REG, aa : a_sub
        sub-rule 6 : src : MEM, dst : MEM, aa : a_add
        sub-rule 7 : src : MEM, dst : MEM, aa : a_sub
```

Figure 7: Example of *and-rule* Flattening

the first tuple X.Y, X denotes an *and-rule* and Y denotes the corresponding *sub-rule*. Rest of the 2-tuples denote the parameters for the string. A 2-tuple (*dot-expression*) corresponding to a parameter is the *dot-expression* associated with the corresponding *and-rule* and *sub-rule*.

In the Sim-nML, instructions are described in a hierarchical manner. The *syntax* and *image* attribute records associated with all the nodes (i.e. *op-rule* and *mode-rule*)

Algorithm 2 :
For each *and-rule*, repeat following steps.

1. Chose[2] an *and-rule* A_i
2. Take the *syntax/image* attribute definitions D of the *and-rule* A_i.
3. For each *sub-rule* S_i of A_1, repeat following steps.
4. If attribute definition D takes no parameter, then D is the resultant string value of *syntax/image* attribute for S_i.
5. If D has reference to a parameter P_i of basic data type, then insert a tuple "$A_i.S_i.P_i$" in D.
6. If D has reference to a parameter P_i of type *and-rule* (say `A`), where number of *syntax/image* attribute values associated with `A`[3] are `n`, then create `n` new *syntax/image* attribute values by substituting each *syntax/image* attribute value in place of parameter reference P_i one by one. These n-new attribute values are associated with the *sub-rule* S_i and so with the *and-rule* A_i.

Figure 8: Algorithm for Flattening of *Syntax/Image* Attribute Definitions

in the specification tree are evaluated. The *syntax* and *image* records of instructions in the instruction set are given by the *syntax* and *image* attribute records of the *op-rule* named `instruction`. Rest of the records hold encoding of partial *syntax* and *image* attribute strings. In the IR, the *syntax* and *image* attribute records for all the *and-rules* are stored. Although tools such as assembler, disassembler, compiler, simulator etc. need only the attribute records of *op-rule* `instruction`, other records might be helpful for other purposes such as to build the specification tree back.

Other attributes in the Sim-nML are used to hold semantic action associated with the instruction. For example, to simulate the behavior of an instruction, attribute definition of *action* attribute is used. A tool such as the instruction set simulator could be made to run faster if such attribute definitions are represented differently. Usually expressions inside an attribute definition are written in an infix notation using priority and associativity rules to decode an expression uniquely. However, prefix or

[2] *And-rules* are chosen by starting with all leaf nodes of specification tree, then all nodes above the leaf nodes and so on.

[3] *Syntax/image* attribute values associated with all *sub-rules* of an *and-rule* are called *syntax/image* attribute values of the *and-rule*.

Before application of algorithm 2 :

```
For and rule 1, mode REG
        sub-rule 0 : i:index
syntax = format("R(%d)",i)
For and rule 2, mode MEM
        sub-rule 0 : i:index
syntax = format("R%d",i)
For and rule 3, op a_add
        sub-rule 0 : no parameter
syntax = "add"
For and rule 4, op a_sub
        sub-rule 0 :  no parameter
syntax = "sub"
For and rule 5, op alu_op
        (see figure 7 for sub-rules)
syntax = format("%s %s %s",aa.syntax,src.syntax,dst.syntax)
```

After application of algorithm 2 :

```
For and rule 1, mode REG
(sub-rule   syntax--string                      dot-expr)
0        "R(%d{1.0.0})"                           "1.0"
For and rule 2, mode MEM
0        "R%d{2.0.0}"                             "1.0"
For and rule 3, op a_add
0        "add"                                    "1.0"
For and rule 4, op a_sub
0        "sub"                                    "1.0"
For and rule 5, op alu_op
0        "add R(%d{1.0.0}) R(%d{1.0.0})"   "5.0(1.0)(1.0)(3.0)"
1        "sub R(%d{1.0.0}) R(%d{1.0.0})"   "5.0(1.0)(1.0)(4.0)"
2        "add R(%d{1.0.0}) R%d{2.0.0}"     "5.0(1.0)(2.0)(3.0)"
3        "sub R(%d{1.0.0}) R%d{2.0.0}"     "5.0(1.0)(2.0)(4.0)"
4        "add R%d{2.0.0} R(%d{1.0.0})"     "5.0(2.0)(1.0)(3.0)"
5        "sub R%d{2.0.0} R(%d{1.0.0})"     "5.0(2.0)(1.0)(4.0)"
6        "add R%d{2.0.0} R%d{2.0.0}"       "5.0(2.0)(2.0)(3.0)"
7        "sub R%d{2.0.0} R%d{2.0.0}"       "5.0(2.0)(2.0)(4.0)"
```

Figure 9: Example of *Syntax* Attribute Definitions Fláttening

In the IR, prefix notation is used for all attribute definitions except *syntax* and *image* attributes. Using such a representation, tools like simulator, trace generator, compiler back-end generator etc. can be made to run fast.

2.2.4 Structure of the Intermediate Representation

As it is evident, the structure of the IR should be capable of storing information about constants, identifiers, *or-rules*, *and-rules* and information about attributes such as *syntax*, *image*, *action* etc. Some of this information can be represented in a fixed size data structure whereas rest of the information requires variable size data structure. For faster retrieval of information, we separate out the variable size data structure and store it at one place.

The IR structure is essentially a collection of various tables. Information of each type is stored in a different table. The entries in most of these tables are fixed size records. However, some tables hold variable size records. We have grouped the similar type of information under same table by creating different record. Also, at some places we created two different tables for clarity although they both hold information in similar type of record. A table of contents is also added in the IR which contains the location and name of all the tables. This simplifies the access mechanism for all tables. In brief, the IR consists of following tables :

- Meta table : This is a table of contents having a road map to know about the location and name of other tables in the IR.
- Constant table : This table holds the all constant declarations in the Sim-nML processor specifications. For the example given in figure 5, this table will contain the following.

```
(name type    value)
REGS  integer 4
```

- Resource table : This table holds the names of the resources which are declared with *resource-declarations*. Each resource is assigned a unique key by which it is referred to at other places.

- Attribute table : This table holds the name and the corresponding key of all distinct attributes used in the input processor specification. For the example given in figure 5, this table will contain the following.

(key	name)
0	syntax
1	image

- Identifier table : This table holds the name of all the identifiers (other than those specified in the constant table and in the resource table). Each identifier is assigned a unique key to refer to the identifier at other places. For the earlier example, the following is the contents of the identifier table.

(key	name	type)
0	PC	reg-var
1	MEM	mode-and
2	REG	mode-and
3	SHORT	mode-or
4	instruction	op-and
5	instr_action	op-or
6	alu_op	op-and
7	move_op	op-or
8	alu_action	op-or
9	a_add	op-and
10	a_sub	op-and
11	move	op-and
12	store	op-and

- Memory table : This table holds the information about all memory variables declared with a *reg* or a *mem* declaration. It includes a unique key, type and size of the data and information to locate various attributes (of the variable) stored in other tables. For the earlier example, the following is the contents of the identifier table.

(key	Name-key	type	size	attribute)
0	0	card(32)	1	-

Note that instead of storing the name of memory variable (i.e. PC), the key assigned in the identifier table is used.

- Or-Rule table : This table holds the information about children of all *or-rules* (*mode-rules* or *or-rules*). It holds records as specified earlier in figure 6.

- And-Rule table : This table holds the information about all *and-rules* (*mode-rules* and *op-rules*) along with the *sub-rules* associated with them. It also holds the information to locate the attribute definitions stored in other tables. It holds records as specified earlier in figure 7.

- Syntax table : This table holds the `syntax-record` associated with the *syntax* attribute definitions of all *and-rules*. It also holds the information to associate the correspondence between the and-rule table and the syntax table as specified earlier in figure 9.

- Image table : This table holds the `image-record` associated with the **image** attribute definitions of all *and-rules*. It also holds the information to associate the correspondence between the and-rule table and the image table. It holds records similar to the syntax table.

- String table : This table is used for storing variable length string (null terminated) such as identifier names. This table helps in having fixed size entries in other tables. For clarity, we used identifier-names and strings in example of tables described earlier. In reality, all such strings are stored in the string table and corresponding index into the string table is stored in other tables.

- Integer table : This table is used for storing only integer values. These integers are associated with other tables and represent different meanings in different contexts. This table helps in having fixed size entries in other table. For example, list of attributes present for an *and-rule* are stored as list of corresponding `attribute-keys` in the integer table. The *and-rule* holds the information which associates the list of integers stored in the integer table as list of `attribute-keys`.

- Prefix-Attribute-Definition Table : This table holds the attribute definition of all the attributes (except *syntax* and *image attributes*) associated with memory-variables and *and-rules*. These definitions are stored in prefix notation. Other

tables store the information to locate the appropriate attribute definition correctly.

In Appendix B, we present the structure of each of the tables. The following two points are important.

1. A crucial decision about the IR is whether it should be a human readable text file or a binary file. We decided to have a binary file as output to enable fast processing by various tools.

2. The data encoding of output file is dependent on the processor on which it is created i.e. data encoding can be little endian or big endian depending on the processor. A tool can figure out the endian-ness of the IR by reading the table of contents irrespective of the type of the machine on which the tools is running. For example, the records of a meta table contain three fields, *no-of-rec*, *size-of-rec* and *size-of-table*. These fields in the first record represent the meta table entries itself. Therefore the *no-of-rec* contains the total number of tables, *size-of-rec* contains the size of each record in the meta table and *size-of-table* contains the total size of the meta table. A tool can read three values and check if the following equation is satisfied.
   *no-of-rec * size-of-rec = size-of-table*
   If this equation is not satisfied, then the endian-ness of the IR and the machine on which the tool is running are not the same, otherwise they are the same.

2.3 Conversion from High Level to Intermediate Representation

The conversion from Sim-nML to the IR is done in the following two passes.

2.3.1 Pass 1 : Macro Preprocessor

The IR does not retain any macro definition from the source. For ease of implementation, macro processing is implemented as a separate pass over Sim-nML specification file. This part is being done in another project by Y. Subhash Chandra[16] but we are

also describing it here for the sake of continuity. The macro preprocessor takes the Sim-nML file with macro definitions as input and produces a Sim-nML file without macros. It gathers all macro definitions and converts them into equivalent m4[18] macro definitions. Then m4, a standard utility available on Unix, is run on this file to get the Sim-nML file without any macros.

2.3.2 Pass 2 : Parsing and Flattening the Hierarchy

Pass two takes a Sim-nML specification file for a processor as input and produces the specification in the IR for that processor. This pass proceeds in three phases.

- The first phase involves the parsing of input file. During the parsing, all relevant information is gathered in appropriate data structures. Attribute definitions for all attributes except *syntax* and *image* attributes are converted into prefix notations during the parsing time. As soon as a definition is complete, it is stored in the prefix-attribute-definition table. In this pass, three temporary files are used to store the string-table, the integer-table and the prefix-attribute-definition table respectively. Each of these table are later merged into the IR.
- In the second phase, first half of the tree flattening is performed. It eliminates references of all *or-rules*.
- In the third phase, second half of the tree flattening is performed. All *and-rules* are flattened further and *syntax* and *image* attributes definition records are created with proper parameter substitutions as described earlier.

At the end of the second pass, all tables are written in the output file and all corresponding data structures are freed. Temporary files generated during this pass are concatenated at proper places in the output file. During this pass, all possible errors at various places are also checked and appropriate error messages are generated. In case of an error in the first phase, the second and the third phases are not performed.

Chapter 3

Design and Implementation of Disassembler

A disassembler is a tool which takes a binary file (relocatable object file, executable file etc.) as input and gives the corresponding assembly language program as output. We have designed and implemented a generic symbolic disassembler (referred to as a disassembler now onwards) which takes an ELF[18] binary file for a processor and generates the assembly language program. The disassembler is generic and processor independent. It takes a processor specification in the IR as another input. The disassembler generates symbols to refer to the locations and functions rather than the absolute addresses in the output assembly language program. Thus the output file resembles the original source from which the binary file was produced. In the output file, the format of assembler directives is the AT&T format[20] and that of the assembly language instructions is the one specified in the processor specification.

The process of disassembly involves reading a binary instruction, searching in the instruction set and generating assembly language instruction. The input binary file contains almost all (*well most of*) the necessary information of the original source file. Unfortunately, the process of disassembly is non-trivial as the binary file is not designed to undergo disassembly. Assemblers throw away a lot of information present in the original source which is irrelevant to the execution of the program. The greatest problem in disassembling is to identify and distinguish code (instructions) and data, as both are represented as sequence of bytes. Furthermore designing a generic disassembler involves extra effort because information about instruction set

of a processor is coded in the processor specification file. Instruction set of the processor must be extracted in a format so that an instruction read from the binary file can be identified easily. In addition, information about number of instructions in the instruction set, length of an instruction, parameters in an instruction etc. varies from processor to processor. Various different processors evaluate the target address for jump instructions using bits available in the instruction in different ways which affects the design of a disassembler.

Lastly, the complexity of the symbolic disassembler is high because it uses symbols to refer to the locations. While programming, users normally use symbols (*names*) to refer to variables and functions. The compilers usually retain the names of functions (and global variables sometimes) in the compiled binary files. However, symbols corresponding to local variables or locations are not retained. Thus disassembler has to generate new names if not available in the binary file.

In this chapter, we shall describe the algorithm used by the disassembler for the disassembly. Basically the approach adopted is to point out what information is available and how it contributes in the generation of the final output.

3.1 Input Binary File (ELF) Structure

Let us begin by examining the structure of the binary file in ELF format (*which is an input to the disassembler*) as taken from the manual[18].

A file in ELF format always begins with a header (*called the ELF header*) which is in a machine independent format so that it can be read on any processor. This header contains information which helps in interpreting the contents of the rest of the file. Thus the ELF header is the master key to the rest of the information in the file. A binary object file contains information grouped together in logical units called *sections*. There are numerous sections in the object file each dedicated to holding a particular kind of information (*program data, code etc.*). Each section has a *section header* which holds the necessary information to interpret the section. The section headers are collected and placed in a table called the *section header table*. The ELF header contains information to locate this section header table.

The sections which are relevant for the purpose of disassembly can be briefly summarized as follows.

- **".text" section**: This section contains the program code.
- **".data" section**: This section contains the initialized global program data.
- **".rodata" section**: This section contains the initialized global read-only data (for example, constants).
- **".bss" section**: This section contains uninitialized global data.
- **".symtab" section**: This section contains information regarding various symbols used in the program (*functions, global variables etc.*). Type, size and lexemes are the chief pieces of information maintained for each entry. The location (*section:offset pair*) is also stored for each entry.
- **"rel.text" section** or **".rela.text" section** : As the name suggests, this section is the relocation section with respect to the `.text` section. This section contains the information needed by the linker to allow it to fill in values of symbols used in the `.text` section which are only available at the link time. Basically this section provides for a mechanism to associate a given offset in the code with an entry in the symbol table. This information is used in the disassembly to regenerate the symbol names in the output.

3.2 Two Pass Design of Disassembler

In order to generate a full assembly file as output, we need to generate :

- instructions (*preferably using symbolic names for memory and symbol references*) within the `.text` section.
- initialized data (*including size and type information along with symbol names*) from the `.data` section.
- read-only data (*this could be character strings*) from the `.rodata` section.
- the makeup of the `.bss` section (*the section meant for uninitialized data, which does not occupy any space in the object file but whose makeup needs to be preserved because symbols may be defined with respect to it*).
- the assembler directives to glue up the whole output.

The processing of the `.text` section is almost independent from the rest. The `.text` section may contain data and other things apart from the program code. Although compilers do not mix data with code, an assembly language programmer may do so. Apart from this, even simple actions like aligning the code for a new function to the nearest 4-byte boundary, can introduce gaps in the `.text` section.

The assembler fills in these gaps by some random (or irrelevant) value since these locations are never executed. The problem is that there is no way to distinguish data from gaps within instructions. If the normal process of disassembly is allowed to take its own course by treating these gaps as genuine code, the opcode alignment may be destroyed. Once misaligned, there is no way to recover and we may get unreliable disassembly. Thus it is absolutely essential to prevent the processing of such gaps. We do this by identifying the basic blocks in the code section. Each basic block constitutes a valid address range in the `.text` section. This is achieved by making one extra pass of code analysis on the `.text` section. Thus our disassembler is a two pass disassembler.

For a generic disassembler, information about a processor's instruction set such as *syntax* and *image* of instructions must be extracted from the intermediate representation of the processor specification. When some instruction uses a reference which can be a symbol, the disassembler needs to resolve the reference for symbolic disassembly and use the symbol-name in the assembly language output program. Therefore, before discussing about the working of first and second pass, we will discuss about references and then about the information extracted from the IR.

3.2.1 Resolving References

The programmers normally code their applications by defining symbols in the program in various sections (`.text`, `.data` etc.). The basic purpose that these symbols serve is to associate a name with a location in one of the sections. The programmer can then refer to these locations using symbol names.

In the ELF file, there is a relocation section (`.rel.text` or `.rela.text`). This section provides relocation information with respect to the `.text` section in most of the relocatable object files. When the assembler encounters a symbol, say in the `.text` section, it creates an entry in the relocation section which associates the occurrence (*the offset at which the symbol reference occurred in the relocatable object file and not*

the location of the actual symbol itself) with the symbol table entry of the symbol. In some cases the assembler may even associate the occurrence with the symbol table entry of the section with respect to which the symbol is defined and include the offset of the symbol as the addend in the reference location. This primarily happens for static variables whose information is not exported at the link time. Both these cases may arise and need to be handled separately.

Now, when some instruction uses a reference which can be a symbol, the disassembler needs to resolve the occurrence of the reference and use the symbol-name. In the best case, the name used could be the same as the original symbol name. While resolving a reference to a location, we normally proceed to determine if there is an entry in the relocation table corresponding to the occurrence, in which case it is enough to resolve whether the entry refers to an object (*global data item*) or to a section (*global-static item*). In the former case, the name is available in the symbol table itself. In the later case, we need to generate a name for the symbol, but only after ensuring that no other name has already been generated for that symbol.

3.3 Extracting Information from Intermediate Representation

The intermediate representation (IR) of a processor specification provides a lot of information about the processor. For the purpose of disassembly, we need the following information about a processor's instruction set.

- **Syntax and Image** : What is the assembly language *syntax* and corresponding binary *image* for the instructions.
- **Arguments information** : For each instruction, how many arguments are needed, the type and length of each of the arguments, how to decode the arguments and how to present the arguments in the assembly language.
- **Control transfer instructions** : which are the instructions which can transfer control from one place to another. These can be further subdivided as unconditional or conditional jump instructions, unconditional or conditional call instructions (to a procedure) and unconditional or conditional return (from a procedure) instructions.

- **Offset calculation** : For a control transfer instruction, how does a processor encode the address of the next instruction.

For a specific disassembler for a processor, all this information can be hard-coded. However for a generic disassembler. this information must be extracted from the intermediate representation of the processor specification.

3.3.1 Extracting Syntax and Image of instructions

The IR of the processor specification contains *syntax* and *image* records for all the instructions. We extract these corresponding to the `instruction` *op-rule*. These records encode the syntax of an assembly language instruction, corresponding binary image and information about the arguments. Arguments type information is found with the help of the and-rule table.

The *image* record includes a string corresponding to the binary image of the instruction. The string does not hold the binary image of the instruction verbatim. For example, a record for add instruction described in figure 5 has the syntax-string as "add r%d{1.0.0},r%d{2.0.0}" and the image-string as "101%2b{1.0.0}1%2b{2.0.0}". The instruction is 8 bits long and it takes two arguments. Both arguments are represented in 2 bits (are `card(2)` type). If instruction "add r2,r3" is assembled, then its corresponding binary image will be "10110111". Therefore, we should have a way to associate the correspondence between the string stored in the *image* record and the binary image of the instruction read from the input binary file. Further, we should be able to find the value of the arguments used by the instructions.

For this purpose, we evaluate two binary strings, namely `image` and `image-mask`, for each of the *image* record. Basically the `image` can be taken as the string value which results from the bit-wise *and*ing of the binary *image* of the instruction and the `image-mask`. Length of `image` is equal to the instruction's length in bits. The algorithm is given in figure 10. For the example of add instruction, the `image` will be "10100100" and the `image-mask` will be "11100100".

Now it is easy to find out whether a given sequence of bits matches with any of the instruction in the instruction set of the processor. It will be a sequential *and*ing and comparing operations on the instruction set. Moreover, if an instruction is matched, then values of all the arguments can be computed to generate the assembly language

Algorithm 3 :
For each *image* record, repeat the following steps 1,2 and 3.

1. Take the image-string from the *image* record.
2. If a bit (0 or 1) is stored in the image-string,
   then {
   - copy the bit value as it is in the image.
   - copy bit '1' in the image-mask.

   }
3. If a parameter reference such as "%d" is stored in the *image* string,
   then {
   - Find the length L of the parameter.
   - Copy L 0s in the image.
   - Copy L 0s in the image-mask.
   - Note the information about the parameter. It includes position, type, length, *and-rule* number, *sub-rule* number and parameter-number. The last three fields are available in the *image* record as a tuple.

   }

Figure 10: Algorithm for Calculating Mask Values

instruction.

3.3.2 Instruction Matching Algorithm

As we described earlier, we can identify whether a given sequence of bits represent an instruction or not using the sequential *and*ing and comparing algorithm. This algorithm is simple but inefficient. The inefficiencies will be even higher if the instruction set have variable length instructions.

We have designed an efficient algorithm as given in figure 11. This algorithm is based on an observation that instruction set of a processor uses some fixed number of bits for opcode in any instruction. By looking at these bits, all the instructions can be divided uniquely into different categories. All instructions will have same bit

Algorithm 4 :

- Find maximum length l_{max} of the instruction.
- Initialize a `general-mask` `G` of length l_{max} with all 1's.
- Do bitwise *and*ing of string `image` of all the instructions with `G`. At end, the `G` will have the required value.
- Now repeat the following steps for all the instructions.
  1. Do bitwise *and*ing of the `image` with the `G` and call it `R`.
  2. if a bucket is having the `bucket-value` same as the `R`, then store the instruction in the bucket.
  3. Otherwise, create a new bucket and store the instruction in the bucket. Assign `R` as the `bucket-value` for this bucket.
- For each bucket, compute `bucket-mask`. The `bucket-mask` is a string resulting from the bitwise *and*ing of all `image` strings of the instructions stored in the bucket.
- Sort the buckets according to the `bucket-values`.
- Sort the instructions within each bucket according to the `image` string.

Figure 11: Algorithm for Calculating More Mask Values

values for the `opcode` in each category. In each category, again some fixed number of bits differentiate among the instructions. We call these bits as `subcode`. Most of the processors use this two-level of hierarchy in assignment of the `opcode` to the instruction.

In the algorithm, a binary string named `general-mask` is calculated to identify the instruction category. We call these category as different buckets. The `general-mask` will have 1 at bit positions used for opcodes. For example, we will get the `general-mask` value as 0xFC 0x00 0x00 0x00 for PowerPC603 processor[19] that has 32 bit long instruction with first 6 bits as an opcode. The instructions are grouped in buckets. Each bucket is assigned a `bucket-value` which is the binary string resultant from the bit-wise *and*ing of the binary string `image` of the instruction and the `general-mask`. Each bucket is assigned a `bucket-mask` to identify a instruction among the instructions stored in each bucket. The `bucket-mask` has bit value

1 at all those positions which are used for the opcode and the subcode. All buckets and the instructions within each bucket are sorted with respect to the `bucket-value` and the binary string `image` respectively to reduce the searching time.

Now the instruction matching algorithm is described as follows.

- Call given sequence of bits to be identified as `D`.
- Do the bitwise *and*ing of the `D` and `general-mask`.
- Find the bucket `B` where the instruction might be stored. For this purpose, do the binary search with comparison of resultant string and `bucket-value`.
- If no bucket is found, then there is no such instruction.
- Otherwise, do the bitwise *and*ing of the `D` and the `bucket-mask` associated with the bucket `B`.
- Find the instruction `I`. For this, do the binary search with comparison of the resultant string and the `image`.
- If search fails, then there is no such instruction.

3.3.3 Extracting Control Transfer Instruction

In the binary file, there may be gaps in between the instructions due to the alignment constraints. Control of the program execution never reaches to such gaps. The flow of a program is affected by the control transfer instructions. We have divided the control transfer instructions under six categories, namely unconditional and conditional jump instructions, unconditional and conditional call (to a procedure) instructions and lastly unconditional and conditional return (from a procedure) instructions. The process of disassembly takes care of such instructions. An occurance of an instruction of such type is used to identify the address ranges that contains the code. Otherwise, disassembled instructions sequence might be completely wrong. Therefore, we need the information about all such instructions.

Instructions under each category can be found by a simple method. The method is based on the assumption that instructions are described in a hierarchical manner in the processor specification. If a complete instruction specification tree is made, then

instruction of a category can be marked under a subtree i.e. an instruction is put under a particular category if the root node of the corresponding subtree is traversed during flattening of the instruction. If a processor specification is not written in this manner, then a little effort is needed to modify it. One can add an *or-rule* with all the instructions of a category as children nodes of the *or-rule*.

The disassembler takes an identifier name for each category from the user. These identifiers denote nodes of various subtrees associated with various categories. As described earlier, the *syntax* and *image* records of an instruction hold dot-expressions which provide the sequence of nodes traversed during flattening of the instruction. If an instruction belongs to any of these category, then the root node of the category tree must be encoded in the dot-expression. If any of these nodes is found in the dot-expression, then the instruction is put under the corresponding category. Otherwise the instruction is not a control transfer instruction and termed as a simple instruction.

There can be a situation when an instruction belongs to two such subtrees. This will happen if tree of one category is also a subtree of another category of tree. For example, the PowerPC processor does not have any `call` type instruction. It uses `jump` type of instructions itself to transfer the control to a subroutine. It stores a return address in the link register and set some bits to treat the `jump` instruction as a `call` instruction. For such conditions, instructions are matched according to a priority rule. We have assigned the priority to unconditional jump, conditional jump, unconditional call, conditional call, unconditional return and conditional return type of instructions respectively in that order. If an instruction matches under two categories, it is put under the higher priority category.

3.3.4 Evaluation of Next Instruction Address

To identify valid code address ranges, it is necessary to evaluate the target address of the control transfer instructions. A control transfer instruction, either gives the starting address of a new address range or causes the end of current address range. For a control transfer instruction, each processor encodes the address of the next instruction in a different way. For example, the next instruction address for jump type of instructions may be specified relative to the current program counter. The relative offset value is encoded in the instruction itself. The encoding of offset value might be different on different processors. In some cases, the next instruction address can

be determined only at the run time, for example, the case when the next instruction address is taken from a processor register or memory. We are interested in finding out the next instruction address from an instruction image of binary file if possible. If its value can not be determined, we do not use it for the identification of code address ranges.

The intermediate representation of the processor holds the attribute definitions for all the instructions. The attribute definition corresponding to *action* attribute simulates the semantic of an instruction. In the Sim-nML, a register called `PC` has special semantic and normally points to the next-to-be executed instruction. For control-transfer instructions, the attribute definition of *action* attribute must modify the `PC` value in some way. In some processors, there are more than one such special register (such as `oldpc`, `newpc`, `currentpc`).

The disassembler takes a set of identifier names from the user. These are essentially the names of such special purpose registers. We extract the attribute definitions corresponding to *action* attribute for all control-transfer instructions. If we have the binary image of an instruction, we can get the address of the next instruction by simulating the execution of the attribute definition. When a statement modifies the value of the program counter (any of the identifier in set entered by the user), its new value is taken as the address of the next instruction. If the statement requires a value which is unknown, then we can not determine the address of the next instruction.

3.4 Implementation Details of the Disassembler

As we said earlier, the disassembler is a two pass disassembler. In reality, these two passes are made over `.text` section only. While disassembling the `.text` section, information is gathered which aid in disassembly of the other sections. In the first pass, it gathers information like references, list of basic block etc. which is used to produce output in the second pass using symbol name for references. The disassembler proceeds in following phases.

3.4.1 Initialization Phase

As said earlier, the disassembler takes ELF binary file and processor specification in the IR as input. The disassembler does the following tasks in this phase.

- It identifies the data encoding of the host processor using the algorithm given in figure 12.
- It checks the integrity of the IR file by looking for the "META TABLE"(table of contents) at the start.
- It then reads the Meta Table entry and detects the data encoding used in the IR file as described in section 2.2.4. If the data encoding of the host processor is different from that in the IR, then a flag is set to indicate that the data read from the IR file must be converted to proper data encoding before its use.
- The disassembler then extracts the information required for disassembly from the IR file as described in the section 3.3.
- It checks the integrity of the binary file by checking the magic number in the ELF header.
- From the ELF header, it detects the data encoding of the binary file and sets a flag if data encoding differs from the source architecture. This indicates that data read from the binary file must be converted into proper data encoding before its use.
- Lastly, it reads in the information from the binary file to be used for future access. This information which includes things like the ELF header, symbol table etc., is held in appropriate data structures so that all the required information is easily available.

3.4.2 First Pass of Disassembly

In the first pass of disassembly, all basic blocks of the code are identified. The identification process is based on the assumption that there must always be some way (*a path*) to reach the code. If there is no such path (i.e. there is no jump/call

(Assume that the *unsigned character* is 1 byte long and the *unsigned integer* is 4 byte long)

- Take a *unsigned character* pointer P and *unsigned integer* pointer I. Let I and P both points to the same 4-byte structure.
- Store 0 at P, P+1 and P+2. Store 0xFF at P+3.
- If value at I is equal to 0x000000FF,
  then data-encoding of the processor is *big-endian.*
- If value at I is equal to 0xFF000000,
  then data-encoding of the processor is *little-endian.*

Figure 12: Algorithm to Find Data Encoding of the Host Processor

to this code), it shall never get executed and hence we need not worry about it. Since the ELF binary file contains the names of the functions in the symbol table, these are taken as the basic blocks in the beginning. The algorithm then proceeds to trace each one of these one by one in order to discover all possible program paths. A stack is used for storing the unprocessed entry points. An instruction is identified using the instruction matching algorithm described earlier in section 3.3.2. After each instruction matching, instruction buffer pointer is moved ahead according to instruction length of the matched instruction. No attempt is made to interpret the contents of the instructions except that a constant vigil is kept over control transfer instructions.

A control transfer instruction, either gives a new entry point or causes the end of the current trace. If the instruction comes under the category of unconditional return instruction or unconditional jump instruction, then the instruction is the last instruction of the current trace. Otherwise tracing is continued. All control instructions except the instructions coming under the category of unconditional return and conditional return, give a new entry point. The address of the next instruction is found by the approach described in the section 3.3.4.

The information gathered in the first pass is stored for use in the second pass. The disassembler maintains a list of pairs. Each association consists of one entry point in the text section and the corresponding name by which it is referred. The list is built up during this pass. Whenever a control transfer instruction refers to an offset

in the code section, an attempt is made to resolve the reference (*using the approach given in section 3.2.1*). If the reference is not resolved, a new symbol is generated and appended to the list. Future references to the same offset would resolve into this new name. At the end, the information (*about the symbols and the entry points*) are sorted with respect to the address of the entry points.

At the end of this pass, all adjacent basic blocks are merged to form a bigger basic block. The intention is to obtain range of addresses which contain only code and no data/gaps. After obtaining these ranges, the second pass simply processes the regions covered by them. Lastly, all basic blocks are sorted with respect to the starting address to ease the translation to the assembly instructions.

3.4.3 Second Pass of Disassembly

The objective of this pass is to generate assembly language instruction from their binary counterpart. Since the address ranges of valid code have been identified in the first pass, we only need to disassemble the instructions in each address range. The instruction disassembly is carried out in the following steps.

- Symbol Generation : To perform the symbolic disassembly, at the beginning of the disassembly of an instruction, it is checked whether a symbol is associated with the address of the current instruction and if so, the symbol type (function name or just a label) is also extracted. If the symbol refers to a function, further information regarding the type and size of the function is also extracted.

- Instruction Generation : An instruction is matched using the instruction matching algorithm (*as described in section 3.3.2*) and corresponding assembly language instruction is output with appropriate parameters. If the instruction is a control transfer instruction, a symbol is found from the symbol table constructed during the first pass and corresponding symbol name is used in the disassembled instruction. Further, memory references to the `.data`, `.rodata` and other such sections are found. In case of such a reference, we try to resolve the address. The size of the operand and the symbol name is stored for the reference.

This procedure is repeated for each instruction in all address ranges. The gaps within the `.text` section are overlooked for the purpose of text disassembly. One

possibility is to simply ignore the bytes in the gaps and change the current location counter so that it reaches the beginning of the next valid address range. However, it is possible that these gaps contain initialized data (*which are not referenced by the normal methods, for instance using register indirection instead of symbols, otherwise it would have been discovered during the first pass*). In such a case, ignoring them might break the intended equivalence between the relocatable object file and the generated assembly code. Therefore, we simply output bytes in the gaps as data and generate appropriate pseudo-ops to glue the code.

3.4.4 Disassembly of Other Sections

While making the second pass through the `.text` section, information is gathered regarding the references made to the other sections. This information together with the symbol table information is used to disassemble the `.data` section. We simply scan the `.data` section looking for those offsets for which a symbol name is available. At these offsets, the corresponding symbol name is output as a label. Moreover, the length information gathered during the second pass is also output for the data item. At all other offsets, the data is dumped byte by byte.

The `.rodata` section is dealt similarly except that the symbol names are not retained in the binary file. Thus, they need to be generated afresh.

Chapter 4

Results and Conclusion

4.1 Results

We have discussed the design of the intermediate representation. The IR fulfills all the goals which were setup behind the design of the IR. Some advantages of the IR are enumerated below.

- All information which was available in the Sim-nML specification can be retrieved. Moreover, parsing effort needed in other tools to get the required information has been saved.
- All forward references of identifiers have been resolved. Thus multiple passes are not necessarily needed in the tools to resolve the references.
- Most of the tables in the IR contain fixed size records. Thus it is easy and efficient to retrieve the required information from the IR.
- Hierarchy of the information has been flattened while retaining the path of flattening. Therefore, processing needed in other tools is reduced.
- All *syntax* and *image* attribute definitions of the instructions are available collectively in one table. Therefore design of the tools such as assembler, disassembler, simulator etc. is simplified as they need to gather information only from one place.

- In the IR, the attribute definitions for all the attributes except *syntax* and *image* are represented in prefix notation. Therefore, tools such as simulator, trace generator, compiler back-end generator etc. can be made to run fast.

- The IR is flexible enough for further extension. One can add more tables in the IR without any problem.

The tool, IR-Generator, is tested for PowerPC603 processor specification[19]. The IR-Generator is tried on Pentium (little-endian) based Linux machines, DEC-Alpha (little-endian) based DEC machines and UltraSparc (big-endian) based Sun OS machines. The inter-operability among these machines is also tried and found to work.

We have also implemented the generic symbolic disassembler. The disassembler can take the IR for a processor specification and ELF binary for that processor as inputs. The salient features of the disassembler are as follows.

- The disassembler is generic and processor independent.

- The disassembler uses symbols to refer to the locations and functions. Therefore, the output file resembles the original source from which binary file was produced.

The disassembler is tested for PowerPC603 IR generated through the IR-Generator. The disassembler is also tried on the above mentioned architectures. It is also verified that it can take the IR generated on a little-endian processor while running on big-endian processor and vice-versa.

The disassembler is tested for several programs. Some of the test results are given in the table 1. All the C programs are compiled using GNU C cross-compiler for PowerPC603 processor running on Pentium based Linux machines. It is observed that all the corresponding instructions are matching in the source assembly program and output assembly programs except those instructions which are not implemented in the specification. Differences in total line numbers are coming due to unimplemented instructions because corresponding binary images are output byte by byte in different lines in the output assembly program.

Some of the example outputs of the disassembler are given here.

Program Number	No. of Lines in C Program	No. of Lines in Assembly Program	No. of Lines in Output Assembly Program
1	68	202	232
2	171	472	505
3	212	664	740
4	224	889	1059
5	505	2487	2698
6	567	2733	4425
7	671	3118	3468
8	693	4262	4469
9	1245	5366	5827
10	2766	13040	15652

Table 1: Test Results

4.1.1 Example 1 :

The following C program compiled using GNU C cross-compiler for PowerPC603 processor running on Pentium based Linux machines.

```
/* file example1.c */
main()
{ int a,b,c;
        a = 20;
        b = 30;
        c = a + b;
}
```

The compilation results into the follwoing assembly program.

```
        .file   "example1.c"
gcc2_compiled.:
        .section .text
        .align 2
        .globl main
        .type   main,@function
main:
        stwu 1,-32(1)
```

```
        stw 31,28(1)
        mr 31,1
        li 0,20
        stw 0,8(31)
        li 0,30
        stw 0,12(31)
        lwz 0,8(31)
        lwz 9,12(31)
        add 0,0,9
        stw 0,16(31)
.L1:
        lwz 11,0(1)
        lwz 31,-4(11)
        mr 1,11
        blr
```

The result of disassembling the corresponding relocatable file using our disassembler is shown below :

```
        .section .text
        .align 4
        .globl main
        .size   main,60
        .type   main,@function
main:
gcc2_compiled.:
        stwu 1,-32(1)
        stw 31,28(1)
        or 31,1,1
        addi 0,0,20
        stw 0,8(31)
        addi 0,0,30
        stw 0,12(31)
        lwz 0,8(31)
        lwz 9,12(31)
```

```
        add 0,0,9
        stw 0,16(31)
        lwz 11,0(1)
        lwz 31,-4(11)
        or 1,11,11
        bclr 20,0
```

As it evident, the disassembler generates a correct assembly language file with necessary assembler directives. Moreover, the symbolic name of the function "main" is retained together with its type, size and binding information. The unnecessary labels used in the source assembly program have been removed. Instructions such as `or` and `mr` are alias of each other. In the output file, `or` instruction is generated as that was the one specified in the processor specification. The output file is cross-compiled and disassembler is run on the corresponding binary file. The generated output is similar to the original one.

4.1.2 Example 2

Let us now take a more complicated example. The source C program is shown below. The program has a conditional `if` statement that gets compiled to multiple branches.

```
/* file "example2.c" */
main()
{ int a,b,min;
        a = 20;
        b = 30;
        if (a > b) min = b;
        else       min = a;
}
```

The assembly program generated by the cross-compiler is shown below.

```
        .file  "example2.c"
gcc2_compiled.:
```

```
        .section .text
        .align 2
        .globl main
        .type  main,@function
main:
        stwu 1,-32(1)
        stw 31,28(1)
        mr 31,1
        li 0,20
        stw 0,8(31)
        li 0,30
        stw 0,12(31)
        lwz 0,8(31)
        lwz 9,12(31)
        cmpw 1,0,9
        bc 4,5,.L2
        lwz 0,12(31)
        stw 0,16(31)
        b .L3
.L2:
        lwz 0,8(31)
        stw 0,16(31)
.L3:
.L1:
        lwz 11,0(1)
        lwz 31,-4(11)
        mr 1,11
        blr
```

The result of disassembling the corresponding relocatable file using our disassembler is shown below :

```
        .section .text
        .align 4
        .globl main
```

```
        .size main,80
        .type main,@function
main:
gcc2_compiled.:
        stwu 1,-32(1)
        stw 31,28(1)
        or 31,1,1
        addi 0,0,20
        stw 0,8(31)
        addi 0,0,30
        stw 0,12(31)
        lwz 0,8(31)
        lwz 9,12(31)
        cmp 1,0,0,0
        bc 4,5,text0
        lwz 0,12(31)
        stw 0,16(31)
        b text1
text0:
        lwz 0,8(31)
        stw 0,16(31)
text1:
        lwz 11,0(1)
        lwz 31,-4(11)
        or 1,11,11
        bclr 20,0
```

For this example, we can observe that the sequence of instructions generated by the disassembler is almost similar to the original except for a few symbol names which are generated by the disassembler. New labels like `text0`, `text1` have the format <section-name, counter>. These symbols are not there in the binary file as these are considered local and thrown away by the assembler.

4.2 Conclusion

Sim-nML language, an extension of nML machine description formalism, is a simple, elegant and powerful enough to model machine behavior at instruction level. In this thesis, we have discussed an integrated environment where Sim-nML acts as the specification language for processor performance model in a generic way. The integrated environment helps in automatic generation of compiler, assembler, disassembler, instruction set simulator and trace generator.

As part of the thesis work, we have designed an intermediate representation (IR) for processor specification written in Sim-nML language. We have demonstrated how the intermediate representation simplifies the development of various tools such as compiler, assembler, disassembler, instruction set simulator, trace generator etc. We have also developed a tool, IR-Generator, which takes a processor specification written in Sim-nML language and produces the intermediate representation of processor specification. Further, a Generic Symbolic Disassembler is developed which takes an intermediate representation of a processor and a relocatable binary file in ELF format as input and produces an equivalent program in assembly language of the processor. We have also given the test results for PowerPC603 processor. Although these tools are tested only for PowerPC603 processor specifications, their design is generic enough to be used for all type of RISC and CISC processor specifications.

4.3 Future Work and Extensions

There are many things which can be undertaken as an extension of this work.

- Flattening of all Attributes : In the IR, all except *syntax* and *image* attribute definitions are stored without any flattening involving parameter substitution. If attribute definitions of all the attributes can be flattened, then it might further simplify the design of some tools such as compiler back-end generator.

- Support for Other File Format : Currently the disassembler can only accept relocatable object files in the ELF format. It would be nice if it could be extended to understand other format also such as COFF, a.out etc.

Appendix A

Grammar of Sim-nML Language

Convention : We have used following convention in describing the Context Free Grammar (CFG) of Sim-nML language.

- *rule*1 : $X|Y$ means either X or Y is derived from rule1. We have written X and Y in separate lines.
- Keywords are written in small-case letters.
- The start symbol is MachineSpec.
- X_Y means the derivation of Y where X is used as a qualifier. For example, Let_Identifier means an identifier specified in LetDef, Const_Expr means a constant expression, Card_Expr means an expression of card type.
- X_Y_Z means the derivation of Z where X and Y both are used as qualifiers. For example. Card_Const_Expr means a constant expression of card type, Para_Mode_Identifier means an identifier used as a parameter name which is of *mode* type.
- Following are the terminal symbols used in describing the grammar. We have used regular grammar notation here.

```
letter       : [a-zA-Z_]
digit        : [0-9]
bin          : [01]
```

```
hex                 : [0-9a-f]
alpha               : [0-9a-zA-Z_]
Identifier          : {letter} {alpha}*
CARD_CONSTANT       : {digit}+
FIXED_CONSTANT      : {digit}+ . {digit}+
BINARY_CONSTANT     : 0b{bin}+
HEX_CONSTANT        : 0x{hex}+
STRING_CONSTANT     : sequence of characters written in double-quotes (" ")
```

Following is the Context Free Grammar for Sim-nML language.

```
MachineSpec        :
                   | MachineSpec LetDef
                   | MachineSpec MacroDef
                   | MachineSpec TypeSpec
                   | MachineSpec ResourceSpec
                   | MachineSpec ExceptionSpec
                   | MachineSpec MemorySpec
                   | MachineSpec ModeSpec
                   | MachineSpec OpSpec
LetDef             : let  Let_Identifier = Const_Expr
MacroDef           : macro ( Macro_Identifier_List ) = Macro_Expr
TypeSpec           : type Type_Identifier = TypeDef
TypeDef            : bool
                   | int ( Card_Const_Expr )
                   | card ( Card_Const_Expr )
                   | fix ( Card_Const_Expr , Card_Const_Expr )
                   | float ( Card_Const_Expr , Card_Const_Expr )
                   | [ Int_Const_Expr .. Int_Const_Expr ]
                   | enum ( Enum_IdentifierList )
                   | instid_type
IdentifierList     : Identifier
                   | IdentifierList  , Identifier
ResourceSpec       : resource Resource_IdentifierList
ExceptionSpec      : exception Exception_IdentifierList
```

```
MemorySpec          : MemRegPart [ Card_Const_Expr , Type ] OptMemAttrList
MemRegPart          : mem Mem_Identifier
                    | reg Mem_Identifer
Type                : TypeDef
                    | Type_Identifier
OptMemAttrList      :
                    | MemAttrDefList
MemAttrDefList      : MemAttrDef
                    | MemAttrDefList MemAttrDef
MemAttrDef          : volatile = String_Const_Expr
                    | alias = MemLocation
                    | initial = Const_Expr
                    | uses = UsesDef
MemLocation         : Mem_Identifier Const_OptBitOptr
                    | Mem_Identifier  [ Card_Const_Expr ] Const_OptBitOptr
ModeSpec            : mode Mode_Identifier ModeSpecPart
ModeSpecPart        : AndRule OptionModeExpr AttrDefList
                    | OrRule
OptionModeExpr      :
                    | = Expr
OpSpec              : op Op_Identifier OpRulePart
OpRulePart          : AndRule AttrDefList
                    | OrRule
OrRule              : = Or_IdentifierList
AndRule             : ( ParamList )
ParamList           :
                    | ParamListPart
                    | ParamList , ParamListPart
ParamListPart       : ParamIdentifier : ParamType
ParamIdentifier     : Param_Rule_Identifier
                    | Param_Mem_Identifier
ParamType           : Type
                    | Rule_Identifier
Rule_Identifier     : Op_Identifier
```

```
                  | Mode_Identifier
AttrDefList       :
                  | AttrDefList AttrDef
AttrDef           : Attr_Identifier = AttrDefPart
                  | syntax = AttrStringExpr
                  | image = AttrStringExpr
                  | action = { Sequence }
                  | uses = UsesDef
AttrDefPart       : Expr
                  | { Sequence }
UsesDef           : UsesOrSequence
                  | UsesDef , UsesOrSequence
UsesOrSequence    : UsesIfAtom
                  | UsesOrSequence | UsesIfAtom
UsesIfAtom        : UsesCondAtom
                  | if Bool_Expr then UsesIfAtom OptionElseAtom endif
OptionElseAtom    :
                  | else UsesIfAtom
UsesCondAtom      : UsesAndAtom
                  | { Bool_Expr } UsesAndAtom
UsesAndAtom       : UsesActionAtom
                  | UsesAndAtom & UsesActionAtom
UsesActionAtom    : UsesDefAtom
                  | UsesDefAtom : UsesActionAttr OptionalTime
UsesActionAttr    : Attr_Identifier
                  | action
                  | Param_Rule_Identifier . action
                  | Param_Rule_Identifier . Attr_Identifier
OptionalTime      :
                  | # { Card_Expr }
UsesLocation      : Mem_Identifier OptBitOptr
                  | Resource_Identifier
                  | Mem_Identifier [ Card_Expr ] OptBitOptr
UsesDefAtom       : UsesLocation OptionalTime
```

```
                    | Param_Rule_Identifier . uses
                    | ( UsesOrSequence )
AttrStringExpr      : Param_Rule_Identifier . syntax
                    | Param_Rule_Identifier . image
                    | String_Const_Expr
                    | format ( STRING_CONST , FormatIdlist )
FormatIdlist        : FormatId
                    | FormatIdlist , FormatId
FormatId            : Param_Mem_Identifier
                    | Param_Rule_Identifier. image
                    | Param_Rule_Identifier. syntax
Sequence            :
                    | StatementList ;
StatementList       : Statement
                    | StatementList ; Statement
Statement           :
                    | nop
                    | action
                    | Attr_Identifier
                    | Param_Rule_Identifier . action
                    | Param_Rule_Identifier . Attr_Identifier
                    | Location = Expr
                    | CondStatement
                    | FunctionName ( ArgList )
                    | error ( STRING_CONST )
FunctionName        : STRING_CONST
ArgList             :
                    | Expr
                    | ArgList , Expr
OptBitOptr          :
                    | < Card_Expr .. Card_Expr >
Location            : MemLocation
                    | ParaLocation
                    | Location :: Location
```

```
MemLocation          : Mem_Identifier OptBitOptr
                     | Mem_Identifier [ Card_Expr ] OptBitOptr
ParaLocation         : Para_Mem_Identifier OptBitOptr
                     | Para_Mem_Identifier [ Card_Expr ] OptBitOptr
CondStatement        : if Bool_Expr then Sequence OptionalElse endif
                     | switch ( Expr ) { CaseList }
OptionalElse         :
                     | else Sequence
CaseList             : CaseStat
                     | CaseList CaseStat
CaseStat             : CaseOption : Sequence
CaseOption           : case Const_Expr
                     | default
Expr                 : UnconditionalExpr
                     | AttrExpr
                     | if Bool_Expr then Expr OptionElseExpr endif
                     | switch ( Expr ) { CaseExprList }
AttrExpr             : Param_Rule_Identifier . syntax
                     | Param_Rule_Identifier . image
                     | Param_Rule_Identifier . Attr_Identifier
OptionElseExpr       :
                     | else Expr
CaseExprList         : CaseExprStat
                     | CaseExprList CaseExprStat
CaseExprStat         : CaseOption : Expr
UnconditionalExpr : ExprPart
                     | coerce ( Type , Expr )
                     | format ( String_Expr , ArgList )
                     | FunctionName ( ArgList )
ExprPart             : LogAndExpr
                     | ExprPart || LogAndExpr
LogAndExpr           : NotExpr
                     | LogAndExpr && NotExpr
NotExpr              : InclusiveOrExpr
```

```
                     | ! InclusiveOrExpr
InclusiveOrExpr      : ExclusiveOrExpr
                     | InclusiveOrExpr | ExclusiveOrExpr
ExclusiveOrExpr      : AndExpr
                     | ExclusiveOrExpr  ^  AndExpr
AndExpr              : EqualityExpr
                     | AndExpr  &  EqualityExpr
EqualityExpr         : RelationalExpr
                     | EqualityExpr  == RelationalExpr
                     | EqualityExpr  != RelationalExpr
RelationalExpr       : ShiftExpr
                     | RelationalExpr <  ShiftExpr
                     | RelationalExpr >  ShiftExpr
                     | RelationalExpr <= ShiftExpr
                     | RelationalExpr >= ShiftExpr
ShiftExpr            : AddExpr
                     | ShiftExpr <<  AddExpr
                     | ShiftExpr >>  AddExpr
                     | ShiftExpr <<< AddExpr
                     | ShiftExpr >>> AddExpr
AddExpr              : MulExpr
                     | AddExpr  +  MulExpr
                     | AddExpr  -  MulExpr
MulExpr              : PowerExpr
                     | MulExpr  *  PowerExpr
                     | MulExpr  /  PowerExpr
                     | MulExpr  %  PowerExpr
PowerExpr            : SimpleExpr
                     | SimpleExpr ** Card_Expr
SimpleExpr           : ( Expr )
                     | - SimpleExpr
                     | + SimpleExpr
                     | ~ SimpleExpr
                     | LocationOpd
```

```
                      | SimpleOpearand
LocationOpd           : MemLocationOpd
                      | ParamLocationOpd
                      | LocationOpd ::  LocationOpd
MemLocationOpd        : Mem_Identifier [ Card_Expr ] OptBitOptr
                      | Mem_Identifier OptBitOptr
ParamLocationOpd      : Param_Mem_Identifier [ Card_Expr ] OptBitOptr
                      | Param_Mem_Identifier OptBitOptr
SimpleOperand         : FIXED_CONST
                      | CARD_CONST
                      | STRING_CONST
                      | BINARY_CONST
                      | HEX_CONST
```

- Note 1 : Following operators give the boolean result in an expression : $!, \&\&, ||, >, >=, <, <=, ==, !=$.
- Note 2 : We have used the zero value as false and the non-zero value as true for boolean expression.
- Note 3 : For MemSpec, one new attribute, `initial` is defined to store the initial values of a memory variable.

Appendix B

File Format of Intermediate Representation

In this appendix, we will discuss the layout of the file for the intermediate representation. The file consists of various fixed or variable size tables where the name of each table is fixed. A table, named as `meta table`, is always the first table in the file. All other tables can reside anywhere in the file and can be located using the `meta table`. The following are the tables available presently in the IR.

- "META TABLE"
- "CONSTANT TABLE"
- "ATTRIBUTE TABLE"
- "RESOURCE TABLE"
- "IDENTIFIER TABLE"
- "MEMORY TABLE"
- "AND RULE TABLE"
- "OR RULE TABLE"
- "SYNTAX TABLE"

- "IMAGE TABLE"
- "STRING TABLE"
- "INTEGER TABLE"
- "PREFIX ATTR DEF TABLE"

Each table consists of an array of records. Each record in a table constitutes of various fields. For each table, all the fields of first records are written first in the file. Then all the fields of second record are written and so on. We have used the word *record* and *entry* interchangeably. The fields might be stored either in little-endian encoding or big-endian encoding depending on the processor on which the file is created.

- Convention : Each table is described by defining its record format. We have used a C-like struct definition to describe a record. For each record, fields are written from top to bottom in the file. In describing the record, following data types are being used :

`Byte`	=	*unsigned char*
`Word`	=	*unsigned short int*
`Dword`	=	*unsigned int*
`SByte`	=	*signed char*
`SWord`	=	*signed short int*
`SDWord`	=	*signed int*
`String`	=	Null terminated array of *characters*
`Address`	=	`Dword`
`Offset`	=	`Dword`

B.1 Meta Table

The `Meta table` holds the table of contents for all the tables which are present in the file. Each record of the `meta table` stores the information to locate a table. Each record has the following format.

typedef struct {

```
    String  table_name;
    Dword   table_size;
    Address table_offset;
    Dword   total_record;
    Dword   record_size;
} Meta_Record;
```

table_name	:	This field stores the fixed name of a table which is a 32 byte null terminated string. Name of all the tables are written earlier.
table_size	:	This field holds the size (in bytes) of a table.
table_offset	:	This field holds the starting offset (in bytes) of a table in the file.
total_record	:	This field holds the number of record stored in a table. For the `string table`, it holds the value 0.
record_size	:	This field holds the size of a record (in bytes) of a table. If a record for a table is variable in size, then this field contains the value 0.

The data encoding of the IR is dependent on the processor on which it is created i.e. data encoding can be little endian or big endian depending on the processor. A tool can figure out the endian-ness of the IR by reading the table of contents irrespective of the type of the machine on which the tools is running. First record of the table represent the meta table entries itself. Therefore the *no-of-rec* contains the total number of tables including the meta table, *size-of-rec* contains the size of each record in the meta table and *size-of-table* contains the total size of the meta table including the first record. A tool can read these values and check if the following equation is satisfied.

no-of-rec * *size-of-rec* = *size-of-table*

If this equation is not satisfied, then the endian-ness of the IR and the machine on which the tool is running are not the same, otherwise they are the same. In the former case, this equation must be satisfied after the endian-ness conversion of the fields values.

B.2 Constant Table

Each record of the `constant table` holds the informations about the `constants` (see section 2.2.1) in the following format.

typedef struct {
  Offset *id_name*;
  Dword *val_typ*;
  SDword *value*;
} Const Record;

id_name : This field holds the index into the `string table`. As discussed earlier, `string table` holds null terminated strings. Thus this field represents a reference to the constant name.

val_typ : This field indicates type of the value associated with the constant (0 for *integer* type or 1 for a *string* type).

value : If the *val_typ* field represents *integer*, then this field holds the corresponding *signed integer* value. If the *val_typ* field represents *string*, then this field holds the *unsigned integer* index into the `string table` from where a null terminated string value can be retrieved.

B.3 Resource Table

Each entry of this table holds the information about a *resource*. Each *resource* is assigned a unique integer key by which it is referenced at other places. Each record has the following format.

typedef struct {
  Offset *res_name*;
  Dword *res_key*;
} Resource_Record;

res_name : This field holds the index into the `string table`. In the `string table`, the name of the *resource* is stored at this index.

res_key : This field holds the key value assigned to the *resource*.

B.4 Identifier Table

This table holds the informations about all the identifiers used in the processor specification file (other than those specified in the `constant table` and the `resource table`). Each identifier is assigned a unique integer key which is used to refer to the identifier at other places. Each record has the following format.

```
typedef struct {
    Offset id_name;
    Dword  id_typ;
    Dword  id_key;
} Identifier_Record;
```

id_name	:	This field holds an index into the `string table`. The `string table` holds a null terminated string at this index which is the name of the identifier.
id_typ	:	This field indicates the type of the identifier and may have one of the following values.

0	:	Undefined Identifier
1	:	Name of a memory Variable
2	:	Name of an *or-rule* of *mode* type
3	:	Name of an *and-rule* of *mode* type
4	:	Name of an *or-rule* of *op* type.
5	:	Name of an *and-rule* of *op* type.
6	:	Name of an Exception
others	:	Unspecified

id_key : This field holds the key value assigned to the identifier.

B.5 Attribute Table

Each entry of this table holds the name of an *attribute*. Each *attribute* is assigned a unique integer key to refer to it at other places. Each record has the following format.

```
typedef struct {
```

Offset *attr_name*;
Dword *attr_key*;
} Attribute_Record;

attr_name : This field holds an index into the `string table`. The `string table` holds a null terminated string at this index which is the name of the *attribute*.

attr_key : This field holds the key value assigned to the *attribute*.

Note : For *mode* specification, one new attribute ,`_val_`, is defined to store the optional expression associated with =.

B.6 Memory Table

Each entry of this table holds the information about a memory variable specified with *reg* or *mem* specification construct of Sim-nML language. Each record has the following format.

typedef struct {
Dword *id_key*;
Dword *siz*;
Dword *tot_attr*;
Dword *mem_reg*;
Dword *data_typ*;
Dword *value1*;
Dword *value2*;
Dword *attr_list_index*;
} Memory_Record;

id_key : This field stores the key value associated with the identifier name of a memory variable. The key value is assigned in the `identifier table`.

siz : A memory declaration defines a memory base i.e. a set of memory locations accessible under a name and an index. This field specifies the number of such locations.

tot_attr : A memory declaration may also define values for some predefined attributes. This field specifies how many attributes are defined for the memory variable.

mem_reg : This field holds a value 0 if the memory identifier is declared using *Reg* specification. It holds 1 if the memory identifier is declared using *mem* specification. Both type of identifiers are similar in nature except that first type of identifiers refer to processor registers and second type of identifiers refer to memory locations.

data_typ :

value1 :

value2 : A memory location might hold values of different data types. The data type is encoded in a tuple <*data_typ, value1, value2*> First field, *data_typ*, specifies what type of values can be stored in a memory location. Second and third field stores the value according to the *data_typ* field. Table 1 shows the possible values for these field.

attr_list_index : If the *tot_attr* field has a value 0, then this field is ignored and should be 0. Otherwise it specifies an index into the `integer table`. At this index, three integers are stored for each of the *attributes*. Therefore, the total number of integers are 3**total_attr*. Each integer triple indicates <*attr_key, offset, len*> where the *attr_key*, is the key corresponding to *attribute name* assigned in the `attribute table`. The second field of triple, *offset*, is the starting tuple number into the `prefix-attr-def table` where definition of the attribute is stored in prefix notation. Third field of triple, *len*, is the number of tuples for its attribute definition.

B.7 And-Rule Table

This table holds the information about all the *and-rules* (*mode* and *op* type). It includes the information about *sub-rules*[1] and *attributes*. The *sub-rules* of an *and-rule* are numbered from 0 to n and parameters are numbered as 0 to m from left to right. Each record has the following format.

[1]Refer to section 2.2.2

Data Type	*data_typ*	*value1*	*value2*
bool	0	0	0
card(n)	1	n	0
int(n)	2	n	0
fix(n, m)	3	n	m
float(n, m)	4	n	m
range[$n..m$]	5	n	m
enum(id_1...id_m)	6	0	$m-1$

Table 1: Encoding of data types

```
typedef struct {
    Dword and_key;
    Dword id_key;
    Dword total_sub_rule;
    Dword total_para;
    Dword total_attr;
    Dword attr_list_index;
    Dword para_list_index;
}And_Rule_Record;
```

and_key : This field holds an integer which is a unique key assigned to an *and-rule*. This key is used later to refer to the *and-rule*.

id_key : This field holds the key value which is assigned to the identifier name of the *and-rule* in the `identifier table`.

total_sub_rule : This field holds the number of *sub-rules* generated by flattening of the *and-rule*.

total_para : This field holds the number of parameters taken by the *and-rule*.

total_attr : This field specifies the number of attributes defined for the *and-rule*.

attr_list_index : If *total_attr* field has value 0, then this field is ignored and has a value 0, otherwise it specifies an index into the `integer table`. At this index, three integers are stored for each of the attributes. Each integer triple indicates <*attr_key, offset and len*> similar to the one described in the `memory table`. There are two exceptions here. If *attr_key* refers to a *syntax* or *image* attribute, then *offset* field contains the starting index in the `syntax table` or

or the `image table` and *len* field contains the total number of *syntax* or *image* records corresponding to the *and-rule*.

para_list_index : If *total_para* field has value 0, then this field is ignored. Otherwise it specifies an index into the `integer table`. At this index, three integers are stored for each of the parameter. Initially, all parameters triples of first *sub-rule* are written, then all parameter triples of second *sub-rule* are written and so on. Thus if we have `n` *sub-rules* and `m` parameters, then there will be `n*m` such integer triples. Each integer triple indicates <*data_typ, value1, value2*> i.e. the data type of parameter. Table 2 shows possible values for fields of the triples.

Data Type	*data_typ*	*value1*	*value2*
bool	0	0	0
card(n)	1	n	0
int(n)	2	n	0
fix(n, m)	3	n	m
float(n, m)	4	n	m
range[$n..m$]	5	n	m
enum(id_1...id_m)	6	0	$m-1$
and-rule	7	*and_key*	0

Table 2: Parameter Type for *and-rule*

B.8 Or-Rule Table

This table holds the information of all *or-rules* (*mode* or *op* type). Each entry describes the children nodes of an *or-rule*[2]. Each record has the following format.

```
typedef struct {
    Dword or_key;
    Dword id_key;
    Dword total_child;
    Dword child_list_index;
}Or_Rule_Record;
```

[2]Refer to section 2.2.2

or_key : This field holds an integer which is a unique key assigned to an *or-rule*.

id_key : This field holds the key value associated with the identifier name of the *or-rule* in the `identifier table`.

total_child : This field holds the integer number which indicate number of children generated by the flattening procedure for the *or-rule*.

child_list_index : This field holds the index into the `integer table` where a list of *and_key* values are stored. Number of such *and_key* values is given by the value of *total_child*. These *and_key* are uses to refer to the *and-rule* (assigned in the `and-rule table`).

B.9 Syntax Table

This table holds the *syntax records* associated with the *syntax* attribute definition of all *and-rules*. Each record has the following format.

```
typedef struct {
    Dword syn_key;
    Dword dot_expr_len;
    Offset dot_expr_offset;
    Dword syn_expr_len;
    Offset syn_expr_offset;
} Syntax_Record;
```

syn_key : This field holds an integer which is a unique key assigned to a *syntax record*. In the `and-rule table`, the key is used to get the attribute information of *syntax* attribute.

dot_expr_len : This field holds the length of a character string, named as *dot-expression*(Refer to section 2.2.3).

dot_expr_offset : This field holds the offset in bytes into the `string table` where actual *dot-expression* is stored as a sequence of characters.

syn_expr_len : This field holds the length of the character string, named as *syntax-string* of the instruction.

syn_expr_offset : This field holds the offset in bytes into the `string table` where the *syntax-string* is stored as a sequence of characters.

B.10 Image Table

This table holds the *image records* associated with the *image* attribute definition of all *and-rules*. Each record has the following format.

```
typedef struct {
    Dword img_key;
    Dword dot_expr_len;
    Offset dot_expr_offset;
    Dword syn_expr_len;
    Offset img_expr_offset;
} Image_Record;
```

img_key	:	This is the unique integer assigned to each *image record.* In the `and-rule table`, this value is used to get the attribute information of *image* attribute.
dot_expr_len	:	This field holds the length of the character string, named as *dot-expression* (Refer to section 2.2.3).
dot_expr_offset	:	This field holds the offset in bytes into the `string table` where actual *dot-expression* is stored as a sequence of characters.
syn_expr_len	:	This field holds the length of the character string, named as *image-string* of the instruction.
syn_expr_offset	:	This field holds the offset in bytes into the `string table` where the *image-string* is stored as a sequence of characters.

B.11 String Table

This table holds null terminated character sequences, commonly called *strings.* These strings are referred to by an index into the `string table`. The first byte at index zero always contains a *null* character. Similarly, the last byte also contains a *null* character, ensuring *null* termination for all strings. A string whose index is zero specifics either no name or a null name depending on the context. We show one example of the `string table` of size 30 bytes in table 3 and the *strings* associated with various indices in table 4.

null	i	d	e	n	t	i	f	i	e
r	null	P	C	null	null	i	n	s	t
r	u	c	t	i	o	n	null	1	null

Table 3: Example of the String Table

Index	*string*
1	identifier
12	PC
16	instruction
18	struction
0	null

Table 4: Interpretation of the String Table

B.12 Integer Table

This table holds list of *unsigned integer* values (`Dword` type). These integers represent different meanings in different contexts. The integers are referred to by an *index* into the **integer table**. The first entry always stored in this table contains 0. The *index* refers to the starting entry and not the starting offset. The offset can be found by multiplying the index and the the size of `Dword`.

B.13 Prefix-Attribute-Definition Table

This table holds various *attribute* definitions in prefix notation. All attributes except the *syntax* and *image* are converted into the prefix notation and stored in this table. Each item of the prefix expression is stored in the following record of type `Tuple_Record`.

```
typedef struct {
    Word   typ;
    SDword value;
} Tuple_Record;
```

typ : This field holds an integer value to indicate the type of tuple i.e. an *operator tuple* or *operand tuple*. If tuple is of *operand type*, then this field also encodes the type of operand.

value : This field holds a integer value which will be interpreted according to the value of *typ* field.

An attribute definition is stored in the `and-rule` table and in the `memory table` with the starting index into the `prefix-attribute-definition` table and the number of items in the prefix notation of the definition. Table 5 shows the possible values of *typ* field and corresponding interpretation of *value* field. If the *typ* field holds the value 0, then the tuple is *operator tuple*, otherwise the tuple is *operand tuple*. If the tuple is of *operator type*, then *value* field holds an integer which indicates operator name and arity. Table 6 shows all possible values for this field and corresponding arity of the operator.

Type of the tuple	*typ* field	*value* field
Operator	0	operator number (see table 6)
Fixed constant	1	`signed integer` value of operand
Card constant	2	`unsigned integer` value of operand
Binary constant	3	Offset into the `string table`
Hex constant	4	Offset into the `string table`
String constant	5	Offset into the `string table`
Memory variable	6	key of the identifier as assigned in the `memory table`
Attribute type	7	key of the attribute name as assigned in the `attribute table`
Parameter type	8	parameter number (left most is assigned number 0).
Resource type	9	key of the resource name as assigned in the `resource table`
Exception type	10	Key of the identifier as assigned in the `identifier table`

Table 5: Interpretation of the tuple used in Prefix Notation

There are as many operands available as needed for an operator. Since the arity for an operator is fixed, the number of arguments is implicit. For example, an expression $PC = PC + 2$ is $= PC + PC2$ in prefix notation and it has 5 items. The first item is an operator '='. Second is a *memory variable* with value field being the index into the `memory table`. Third item is again an operator '+'. The last field is a *fixed-constant* 2.

value	Name of Operator	Symbol	Arity of Operator
0	Addition	+	Binary
1	Subtraction	-	Binary
2	Multiplication	*	Binary
3	Division	/	Binary
4	MOD	%	Binary
5	EXP	**	Binary
6	Greator than	>	Binary
7	Less than	<	Binary
8	Equal to	==	Binary
9	Not equal to	!=	Binary
10	GEQ	>=	Binary
11	LEQ	<=	Binary
12	Logical AND	&	Binary
13	Logical OR	|	Binary
14	Logical XOR	^	Binary
15	AND	&&	Binary
16	OR	||	Binary
17	Left Shift	<<	Binary
18	Right Shift	>>	Binary
19	Rotate Left	<<<	Binary
20	Rotate Right	>>>	Binary
21	Dot	.	Binary
22	Concatenation	::	Binary
23	Indexing	[]	Binary
24	Assignment	=	Binary
25	Statement Separator	;	Binary
26	Unary Addition	+	Unary
27	UNOT OPERATOR	!	Unary
28	Unary Subtraction	-	Unary
29	Bitwise NOT	~	Unary
30	Bit Range	..	Ternary
31	IF	if then else	Ternary
32	Function	canonical function	n-ary
33	Switch	switch	n-ary
34	default Expression	default	0-ary
35	NULL	nothing	0-ary
36	Hash	#	Binary
37	Comma	,	Binary
38	Condition	{}	Unary
39	Colon	:	Binary

Table 6: Operators Used in Prefix Attribute Definition

For detailed description of each operator, read the Sim-nML specification given in Appendix A. There are some special cases which are described here.

- The first case is for Bit Range operator which has the infix notation as $opd1 < opd2..opd3 >$.
  Equivalent prefix notation used is as follows.
  $(operator, bitrangeoperator, opd1, opd2, opd3)$.

- The second case is for "if then else". If there is no operand in *else* part, then NULL operator (0-ary) (see table 6) is being used.

- The third case is when there is a no attribute expression for an attribute. We have used NULL operator to denote it.

- The fourth case is that of a `switch` operator. General infix notation for this is

```
switch (expr)
{
   case   Expr_1  :  Sequence_1 ;
   case   Expr_2  :  Sequence_2 ;
   .
   default        :  Sequence_i ;
   .
   case   Expr_n  :  Sequence_n ;
}
```

The corresponding pre-fix notation is as follows :

```
(operator, switch)
   (n, expr,
       Expr_1,            Sequence_1,
       Expr_2,            Sequence_2,
       ....
       DEFAULT OPERATOR, Sequence_i,
       ....
       Expr_n,           Sequence_n)
```

The first item is an operator with operator name as `switch`. Then next item is a simple operand tuple of Card constant type and value as n. After that, expr will be again written in prefix notation. It will be followed by n-operands where each operand is an expression in prefix notation and sequence of statements in prefix notation. Default operator is a 0-ary operator so it can be taken as a pre-fix expression.

- The fifth case is that of a canonical function. General notation for this is as follows.
  "function name" $(Arg1, Arg2, Arg3, \ldots\ldots\ldots, Argn)$
  where each argument is again an expression. The corresponding pre-fix notation is as follows.

```
(operator, function)
  (length of name, "function name" string,
   n, Arg1, Arg2,........Argn)
```

The first item is a function operator. Second tuple is a string constant type (`typ` = String constant, `value` = byte offset into the string table where function name is written). Next item is a simple operand tuple with `typ` as Card constant and `value` as n. Then each argument is represented in prefix notation.

There is one special case with function operator where the function name is *coerce*. This function takes first argument as a data type. In the IR, we convert data types to the basic data types and represent them using three numbers, *data_type, value1* and *value2* as described in table 1. Thus, the data type parameter for the *coerce* function is converted to three integers internally. Therefore, we have two extra parameters for this function. Thus number of parameters are increased by two.

Appendix C

User's Manual

In this thesis, we developed two tools, IR-Generator and disassembler. Both the tools have a command line interface that is conventional for the utilities/commands in a Unix system. If a tool is run without any arguments, then it displays a small help giving all the options.

C.1 IR-Generator

The IR-Generator is used to translate Sim-nML specification into the IR. It is available as a command called 'irg'.

C.1.1 Usage

```
Use : irg [-d] [-h] [-w] [-o ir_file_name] Sim_nML_input_file
 -d : To get debug info in debug.tmp
 -h : to get this message
 -w : to get warning messages. Default no warning
 -o ir_file_name : IR will be in file ir_file_name otherwise
      default file name is IR.
```

Descriptions for all options are as follows.

- -d : This is an optional argument. It makes available a lot of debugging information in the "debug.tmp" file in the current directory. By default, no debugging information is generated.
- -h : This is an optional argument. If this option is specified, then a small help message is generated and all other arguments are ignored.
- -w : This option is used to see the warning messages. These message are generated while translating the Sim-nML specification file into the IR. By default, no warning messages are displayed. These messages provide the information which might be useful for the specification writers. For example, *action* attribute for all *and-rules* must be specified. If there is an *and-rule* with no *action* attribute definition, then a warning is displayed. Sometime this may be the intention of the user while sometime this may be an error.
- -o *ir_file_name* : This option is used to set the output file name. By default, output file named "IR" is created in the current directory.
- *Sim-nML_input_file :* This argument must always be present which represents the input Sim-nML file without *macros*.

If some errors occur in the translation, then appropriate error messages are displayed and tool exits.

C.2 Disassembler

The disassembler is used to translate a relocatable binary code to its assembly language counterpart. The input binary file must be in the ELF format. The disassembler also requires a processor specification in the IR. The disassembler is available as a command 'disa'.

C.2.1 Usage

```
Use : disa [-d] [-h] -w] [-o output_file_name] [-i ir_input_file]
          [-c config_file] objfile_name
 -d : To get debug info in debug.tmp
```

```
-h : to get this message
-w : to get warning messages. Default no warning
-o output_file_name : output assembly language file name
   otherwise default file name is outfile.s
-i ir_file_name : input file having IR of processor specification
   otherwise default file name is IR
-c config_file : input file having various arguments for Disassembler
   otherwise default file name is CONFIG
obj_file_name : input relocatable ELF file to be disassembled
```

Descriptions for all options are as follows.

- -d : This is an optional argument. It makes available a lot of debugging information in the "debug.tmp" file in the current directory. By default, no debugging information is generated.

- -h : This is an optional argument. If this option is specified, then a small help message is generated and all other arguments are ignored.

- -w : This option is used to see the warning messages. These message are generated while disassembling the binary file. By default, no warning messages are displayed.

- -o *output_file_name* : This option is used to name the output file containing the assembly language program. By default, output file is named "outfile.s" and is created in the current directory.

- -i *ir_file_name* : This option is used to name the processor specification file in the IR. By default, input file named "IR" is used in the current directory.

- -c *config_file* : This option is used to name the configuration file. This file contains the list of identifiers corresponding to control transfer instructions. By default, input file named "CONFIG" in the current directory is used as configuration file. The format of each line in the configuration file is as follows.

  ```
  %identifier_type identifier_name
  ```

  Each line starts with "%" followed by a type name. This type name provides the type of identifier followed and can be one of the following.

- BRANCH_UNCOND
- BRANCH_COND
- CALL_UNCOND
- CALL_COND
- RETURN_UNCOND
- RETURN_COND

After the *identifier_type*, name of an identifier is followed. Basically, each identifier name corresponds to the category of control transfer instructions given by *identifier_type*[1] and represents a *sub-tree* for its class in the specification file.

The *config_file* also holds the names of identifiers which are used as program counter in the processor specification. Format for specifying this information is similar except that a list of identifier names can be given separated by commas. The *identifier_type* field will have the value PC_CLASS. All these lines can be in any order. An example file is given in figure 13 for more clarity. Here CIA and NIA are the names of the PC_CLASS registers used in the specification file.

```
%PC_CLASS CIA, NIA
%BRANCH_UNCOND branch_uncond
%BRANCH_COND branch_cond
%CALL_UNCOND call_uncond
%CALL_COND call_cond
%RETURN_UNCOND bran_cond_lr
%RETURN_COND ret_cond
```

Figure 13: Example of the Configuration File

- *obj_file_name* : This mandatory option provide the ELF binary file name to be disassembled.

[1]Refer to section 3.3.3

Bibliography

[1] FREERICK, M. The nML Machine Description Formalism, July 1993. *http://www.cs.tu-berlin.de/~mfx/dvi_docs/nml_2.dvi.gz.*

[2] GUILFANOV, I. IDA-Pro : The Multi-Processor Multi-OS Interactive Disassembler. http://www.datarescue.com/ida.htm.

[3] MATURANA, JAMES, AND JEFFERY. A Cycle Accurate Model of UltraSPARC. In *Proceedings of the 1995 International Conference on Computer Design (ICCD '95)* (1995). http://www.computer.org/conferen/proceed/iccd95/abstract.htm.

[4] MOONA, R., AND V.RAJESH. Processor Modeling for Hardware-Software Codesign. *International Conf. on VLSI Design* (Jan 1999).

[5] MSLEE. Processor Modeling and Verification, May 1997. http://camars.kaist.ac.kr/~mslee/abstract/mod_ver.html.

[6] RAKSEY, N., AND FERNANDEZ. Specifying Representations of Machine Instructions. *ACM Transaction on Programming Langauges and Systems 19* (May 1997). http://www.cs.virginia.edu/~nr/pubs/specifying-abstract.html, http://www.cs.virginia.edu/~nr/toolkit/examples/sparc/sparcdis.html.

[7] RAMSEY, N., AND CIFUENTES, C. SPARC Disassembler. http://www.cs.virginia.edu/~nr/toolkit/examples/sparc/sparcdis.html.

[8] ROSE, STEEVES, AND CARPENTER. VHDL Performance Modelling, Aug 1997. http://www.htc.honeywell.com/projects/rassp/RASSP94/conf94_1.html.

[9] Processor Models. References to VHDL Tools : http://rassp.scra.org /vhdl/models/modsl_quick_index.html.

[10] SHEN, J. P., AND NOONBURG, D. B. A Framework for Statistical Modeling of Superscalar Processor Performance. *Third IEEE Symposium on High-Performance Computer Architecture (HPCA-3)* (1997). http://www.foolabs.com/derekn/.

[11] Trimaran : An Infrastructure for Research in Instruction-Level Parallelism, Sep 1998. http://www.trimaran.org.

[12] MDES Manual. http://www.trimaran.org/docs/mdes_manual.pdf.

[13] TRUNG A., D., AND JOHN PAUL, S. VMW: A Visualization-Based Microarchitecture Workbench. *IEEE Computer* (Dec 1995), 57–64.

[14] V.RAJESH. A Generic Approach to Performance Modeling and its Application to Simulator Generator. Master's thesis, Department of Computer Science and Eng., IIT Kanpur, July 1998. *http://www.cse.iitk.ac.in/users/vrajesh/simnml.*

[15] WOLF, AND YEN. Performance Estimation for Real- Time Distributed Embedded Systems. In *Proceedings of the 1995 International Conference on Computer Design (ICCD '95)* (1995). http://www.computer.org/conferen/proceed/iccd95/abstract.htm.

[16] Y. SUBHASH, C. M.Tech. Thesis Work, yet to be submitted. Master's thesis, Department of Computer Science and Eng., IIT Kanpur, April 1999.

[17] Disassembling. References to various Disassembler : http://www.it.uq.edu.au /MENU/RESEARCH_GROUPS/csm/decompilation/ disasm.html.

[18] *UNIX System V Release 4, Programmers Guide : ANSI C and Programming Support Tools*, 1992.

[19] *PowerPC 603 RISC Microprocessor User's Manual*, 1994.

[20] GNU Assembler Manual. http://www.freebsd.org/info/as-all/as-all.info. Manual.html.